НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

До захисту допущено:

Завідувач кафедри

_____ Сергій СТІРЕНКО

«___»_____20__ р.

Дипломний проєкт

на здобуття ступеня бакалавра

за освітньо-професійною програмою «Інженерія програмного забезпечення комп'ютерних систем»

спеціальності 121 «Інженерія програмного забезпечення»

на тему: «Система зберігання даних в пам'яті»

Виконав (-ла):
студент IV курсу, групи ІП-64
Граб Ярослав Олегович _____

Керівник:
доцент, к.т.н.
Волокита Артем Миколайович _____

Консультант з нормо контролю:
Професор, доктор технічних наук
Сімоненко Валерій Павлович _____

Рецензент:

_____

Засвідчую, що у цьому дипломному проєкті немає запозичень з праць інших авторів без відповідних посилань.

Студент (-ка) _____

Київ – 2020 року

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
ІМ. ІГОРЯ СІКОРСЬКОГО»

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

Рівень вищої освіти – перший (бакалаврський)

Спеціальність – 121 «Інженерія програмного забезпечення»

Освітньо-професійна програма «Інженерія програмного забезпечення комп'ютерних систем»

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ *Сергій СТІРЕНКО*
(підпис)
"___"_____20__ р.

ЗАВДАННЯ

на дипломний проєкт студенту

Грабу Ярославу Олеговичу

1. Тема проєкту «*Система зберігання даних в пам'яті»*

керівник проєкту Волокита Артем Миколайович, доцент, к.т.н., затверджені наказом по університету від *«07»* травня 2020р. №__1081-с__

2. Термін здачі студентом закінченого роботи *27 травня 2020р.*
3. Вихідні дані до проекту*: технічна документація, теоретичні та статистичні дані.*
4. Зміст пояснювальної записки:

   Розділ 1. Аналіз існуючих рішень

   Розділ 2. Огляд архітектури системи

   Розділ 3. Опис програмної реалізації

Розділ 4. Результати дослідної експлуатації

5. Консультант роботи, з вказівкою розділів роботи, які до них вносяться

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв
Розділ 1	доцент, к.т.н. Волокита А. М.		
Розділ 2	доцент, к.т.н. Волокита А. М.		
Розділ 3	доцент, к.т.н. Волокита А. М.		
Розділ 4	доцент, к.т.н. Волокита А. М.		

6. Дата видачі завдання *14.12 .2019 року*

КАЛЕНДАРНИЙ ПЛАН

п/п	Найменування етапів дипломного проєкту (роботи)	Строк виконання етапів проєкту(роботи)	Примітки
1.	Затвердження теми роботи	14.12.2019-19.12.2019	Виконано
2.	Вивчення та аналіз завдання	19.12.2019-19.03.2020	Виконано
3.	Розробка архітектури та загальної структури системи	19.03.2020-30.03.2020	Виконано
4.	Розробка структур окремих підсистем	30.03.2020-10.04.2020	Виконано
5.	Програмна реалізація системи	10.04.2020-20.04.2020	Виконано
6.	Оформлення пояснювальної записки	15.04.2020-10.05.2020	Виконано
7.	Захист програмного продукту	25.04.2020	Виконано
8.	Перед захист	28.05.2020	Виконано
9.	Захист	19.06.2020	Виконано

Студент Ярослав ГРАБ_____

(підпис)

Керівник Артем ВОЛОКИТА_____

(підпис)

АНОТАЦІЯ

У роботі розроблено клієнт-серверний програмний продукт для зберігання даних в оперативній пам’яті. Розглянута архітектура клієнтської та серверної частин, описані технології та інструменти, необхідні для їх програмної реалізації. Описано інтерфейс та основний функціонал усіх структурних компонентів.

Під час створення системи були враховані усі вимоги, що повинні бути враховані при проектуванні подібних систем. Оптимально спроектована архітектура додатку забезпечила високу швидкість роботи та безпеку даних, що перебувають у сховищі.

Розроблене програмне забезпечення має простий та ефективний користувацький інтерфейс, що дозволяє дуже чітко виконувати поставлені задачі.

Ключові слова: IMDG, сховище даних в оперативній пам’яті, кластер, клієнт, сервер, балансування, відмовостійкість.

ABSTRACT

There was developed a client-server software product for storing data in RAM. The architecture of client and server parts is considered, the technologies and tools necessary for their software implementation are described. The interface and the main functionality of all structural components are described.

During the creation of the system, all the requirements that must be taken into account when designing such systems were taken into account. The optimally designed architecture of the application ensured high speed and security of stored data.

The developed software has a simple and effective interface, which allows to perform the tasks very accurately.

Keywords: IMDG, in-memory storage, cluster, client, server, data-balancing, fault tolerance.

ВІДОМІСТЬ ДИПЛОМНОГО ПРОЄКТУ

з/п	Формат	Позначення	Найменування	Кількість листів	Примітка
	А4		Завдання на дипломний проєкт		
	А4	ІАЛЦ.467100.001 ВП	Відомість проєкту	1	
	А4	ІАЛЦ.467100.002 ТЗ	Технічне завдання	3	
	А4	ІАЛЦ.467100.003 ПЗ	Пояснювальна записка	58	
	А3	ІАЛЦ.467100.004 Д1	Алгоритм дій програмного забезпечення	1	
	А3	ІАЛЦ.467100.005 Д2	Функціональна схема(діаграма класів)	1	
	А3	ІАЛЦ.467100.006 Д3	Структурна схема веб-застосунку	1	
	А4	ІАЛЦ.467100.007 Д4	Текст програмного коду	26	

					ІАЛЦ.467100.001 ВП		
Зм.	Арк.	№ документна	Підпис	Дата			
Розробив		Граб Я.О			Система зберігання даних в пам’яті Відомість дипломного проекту	Літ.	Аркуш 1 / Акрушів 1
Перевірив		Волокита А.М.					
Реценз.						НТУУ КПІ ім. Ігоря Сікорського, ФІОТ, ІП-64	
Н. Контр.		Сімоненко В.П.					
Затверд.							

ТЕХНІЧНЕ ЗАВДАННЯ
ДО ДИПЛОМНОГО ПРОЕКТУ

на тему: «Система зберігання даних в пам’яті»

Київ – 2020

ЗМІСТ

1 НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ

Дане технічне завдання стосується розробки системи зберігання даних в оперативній пам’яті, яка повинна бути відмовостійкою, масштабованою, здатною зберігати різні типи даних.

2 ПІДСТАВИ ДЛЯ РОЗРОБКИ

Підставою для розробки є завдання на виконання бакалаврської дипломної роботи, затверджене кафедрою обчислювальної техніки Національного технічного університету України «Київський політехнічний інститут» імені Ігоря Сікорського.

3 МЕТА ТА ПРИЗНАЧЕННЯ РОЗРОБКИ

Метою розробки є створення системи зберігання даних в оперативній пам’яті.

4 ДЖЕРЕЛА РОЗРОБКИ

Джерелами розробки є науково-технічна література, монографії, публікації в періодичних виданнях та мережі Інтернет.

5 ТЕХНІЧНІ ВИМОГИ

5.1. Вимоги до програмного продукту, що розробляється

Система, що розробляється, повинна:

1) зберігати різні типи даних;
2) реалізовувати механізм масштабування;
3) бути відмовостійкою;

5.2. Вимоги до інструментального програмного забезпечення

– встановлене середовище виконання .NET Core 3.0;

5.3. Вимоги до апаратної частини обчислювальної системи

– ОС: MS Windows 7/8/10, 32/64-bit;

– об’єм оперативної пам’яті: 2 Гб;

– об’єм вільного простору на жорсткому диску: 100 Мб;

– тактова частота процесора: 1,4 ГГц і вище;

6 ЕТАПИ РОЗРОБКИ

Назва етапів виконання	Термін виконання
Затвердження теми роботи	
Вивчення та аналіз завдання	
Розробка архітектури та загальної структури системи	
Розробка структур окремих частин системи	
Програмна реалізація системи	
Виправлення помилок	
Оформлення пояснювальної записки	
Передзахист	
Захист	

ПОЯСНЮВАЛЬНА ЗАПИСКА
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «Система зберігання даних в пам’яті»

Київ – 2020

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

IMDG – розподілена система зберігання даних в оперативній пам’яті.

Кластер – група незалежних обчислювальних машин, що використовуються спільно і працюють як одна система.

СУБД – система управління базами даних.

ACID – набір властивостей, які гарантують надійність роботи транзакцій баз даних: атомарність, узгодженість, ізольованість, довговічність. Набір операцій над базою даних, що відповідає ACID властивостям, можна розглядати як одну логічну операцію над даними.

InfiniBand – високошвидкісна комутована послідовна шина, яка використовується для внутрішньосистемних та міжсистемних з'єднань. Описи InfiniBand є специфікованими. Розвитком і підтримкою специфікацій займається InfiniBand Trade Association.

NoSQL – підхід до проектування баз данних, при якому дані пов’язані не реляційними відносинами (доступ по ключу, графи, документи тощо).

SQL – Structured Query Language, мова запитів, яка надає можливість створювати запити до реляційної бази даних.

OQL – Object Query Language, об'єктно орієнтована мова запитів. Являється об'єктним розширенням мови SQL, що надає можливість формувати запити як до реляційної, так і до об’єктної бази даних.

NewSQL – клас реляційних баз даних, завданням яких є об’єднати переваги NoSQL і класичних баз даних.

IMDB – резидентна база даних, розміщена в оперативній пам’яті.

ВСТУП

У наш час обробка інформації в оперативній пам’яті є надзвичайно популярною тематикою, особливо зважаючи на той факт, що чимала кількість компаній, які раніше відмовлялися розглядати використання in-memory технологій через високу вартість, зараз перебудовують архітектуру своїх інформаційних систем, щоб використовувати переваги швидкої транзакційної обробки даних, пропонованих даними рішеннями. Це є наслідком стрімкого падіння вартості оперативної пам'яті (RAM). Порівняно із обробкою даних на жорстких дисках, наслідком цього факту стає можливість зберігання всього набору операційних даних в пам'яті зі збільшенням швидкості їх обробки більш ніж в 1000 разів. Для реалізації таких рішень In-Memory Compute Grid та In-Memory Data Grid продукти мають усі необхідні інструменти.

Зазначений підхід дуже швидко набув широкого визнання серед експертів. У першу чергу він зацікавив спеціалістів у галузі проектування хмарних платформ. Одночасно він актуальний і серед дослідників будь-яких систем, залежних від практично необмеженого масштабування системи зберігання даних. Багато відомих компаній випустили на ринок системи такого типу. Проте, більшість таких програмних продуктів є і надто дорогими для практичного використання, і мають обмежену кількість.

Основними перевагами розробленої системи є відкритий доступ, можливість зберігання в оперативній пам’яті складних структур даних, вбудовані реплікація даних та балансування. Саме це і стало причиною розробки даної комп’ютерної системи.

Після встановлення системи користувач матиме можливість додавати, редагувати, видаляти, отримувати дані зі сховища. Крім того, передбачена можливість роботи зі складними структурами даних.

РОЗДІЛ 1
АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ

1.1 Теоретичний аналіз In-Memory Data Grid (IMDG) реалізації розподілених сховищ даних

В останнє десятиліття широкого поширення набули розподілені сховища даних в оперативній пам'яті (in-memory data grid, IMDG). Така ситуація пов’язана з постійним підвищенням вимог до швидкості доступу до даних.

Початкова концепція таких сховищ мала за мету збір даних із IMDG тільки методом доступу через ключ (т.зв. NoSQL підхід). Одночасно передбачили можливість здійснення призначених для користувача функцій одразу на серверах сховища. Проте завдяки досвіду застосування цих сховищ була виявлена необхідність виконання SQL-подібних запитів. На цей час в найбільш поширених реалізаціях IMDG підтримується об'єктна мова запитів (object query language, OQL), хоча й це відчувається лише частково.

У порівнянні з обробкою даних в оперативній пам'яті, сучасні технології передачі даних можуть виконувати свої функції повільніше. Мережа, побудована за технологією InfiniBand з номінальною пропускною спроможністю 40 Гбіт / сек., показала можливі результати. Зокрема 10 Гбіт / сек на вузлі у процесі секціонування даних за допомогою хешфункції. У ході виконання такої опції в оперативній пам'яті, секціонування на кілька тисяч частин виконується з великою швидкістю. Вона приблизно рівна пропускній здатності оперативної пам'яті [1, 2].

Поняття «грід» тривалий час означало добровільне об'єднання обчислювальних ресурсів. А також його ідентифікувати із середовищем виконання наукових розрахунків. Однак сьогодні стало очевидно, що об'єднання в єдину розподілену інфраструктуру ресурсів пам'яті серверів, що

входять в кластер, забезпечує прискорення роботи в режимі реального часу. Особливо зазначене стосується структури пам'яті великих обсягів.

Завданням In-Memory Data Grid (IMDG) є забезпечення надвисокої доступності даних. Це завдання може бути реалізованим при допомозі зберігання їх в оперативній пам'яті в розподіленому стані. Сучасним IMDG під силу задоволення більшості вимог до обробки великих обсягів даних.

Розробку in-memory СУБД вперше було розпочато в 1993 році в Bell Labs., тоді ж система була прототипована як Dali Main-Memory Storage Manager. Цими дослідженнями закладено початок створення першого комерційного виробу in-memory СУБД – Datablitz.

Найбільші гравці ринку баз даних у наступні роки звернули увагу на in-memory СУБД. Наприклад TimesTen, компанію-стартап, засновану Марі-Енн Неймат (Marie-Anne Neimat) в 1996 році як відгалуження від Hewlett-Packard, у 2005 році придбала Oracle. Сьогодні зазначена компанія продає цей продукт, так само як і in-memory СУБД. У 2008 році IBM купила SolidDB in 2008. Ця компанія також здійснює роботу у сфері in-memory СУБД і Microsoft.

VoltDB, яку заснував один із піонерів ринку СУБД Майкл Стоунбрейкер (Michael Stonebraker), анонсувала вихід in-memory СУБД. Ця процедура відбулася в травні 2010 року, а сьогодні ж компанія пропонує і вільну, і пропрієтарну версію цієї системи. SAP в червні 2011 року випустила in-memory СУБД, SAP HANA.

Розподілені сховища даних в оперативній пам’яті (In-memory data grid, IMDG) сьогодні набувать значної популярності, висуваючи головним завданням цієї технології скорочення часу доступу до даних. Ця дія стає можливою саме завдяки їх збереженню через оперативний запам’ятовуючий пристрій (ОЗП) в розподіленому стані [1].

Спрощено, IMDG - це розподілене сховище об'єктів. За інтерфейсом воно схоже зі звичайною багатопотоковою хеш-таблицею, у якій користувач зберігаєте об'єкти за ключами. Однак, на відміну від традиційних систем, в яких

ключі і значення обмежені типами даних «масив байт» і «рядок», в IMDG користувач використовує будь-який об'єкт із Вашої бізнес-моделі як ключ або значення. Ця операція суттєво підвищує гнучкість і дозволяє користувачеві зберігати в Data Grid виключно той об'єкт, з яким працює бізнес-логіка. Все це відбувається без додаткової серіалізації / десеріалізації, які необхідні для альтернативних технологій. Зазначене також спростить використання IMDG систем, тому користувач як зі звичайною хеш-таблицею зможете працювати і з розподіленим сховищем даних.

Одна з основних відмінностей IMDG від In-Memory баз даних (IMDB) - можливість працювати з об'єктами з бізнес-моделі безпосередньо. У цьому випадку користувачам ще доведеться здійснювати об'єктно-реляційне відображення (Object-To-Relational Mapping). За таких обставин варто пам’ятати, що воно переважно призводить до значного зниження продуктивності.

Існують й альтернативні функціональні особливості. Вони відрізняють IMDG від інших продуктів, зокрема, таких як IMDB, NoSQL або NewSQL бази даних. Одна з основних – дійсно масштабується секціонування даних (Data Partitioning) в кластері, що відображено на рисунку 1.1.

IMDG в дійсності становить собою розподілену хеш-таблицю, де кожен ключ зберігається на конкретному сервері в кластері. У більшому кластері можна зберігати більше даних. Принципово актуальним у цій архітектурі є те, що обробку даних потрібно проводити на тому ж сервері, де вони розташовані (локально). Слід виключити (або звести до мінімуму) їх переміщення по кластеру.

Отже, в умовах використання добре спроектованого IMDG, переміщення даних буде повністю відсутнє. Вийнятком може стати випадок, коли в кластер додаються нові сервери або видаляються існуючі. Внаслідк цього може статися зміна топології кластера і розподіл даних в ньому.

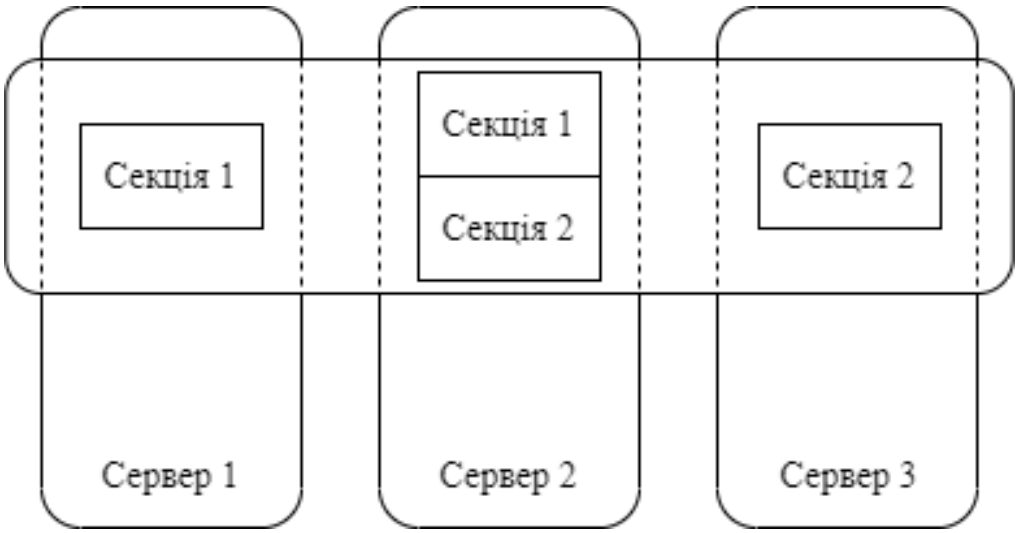


Рисунок 1.1. Секціонування в розподілених сховищах

До основних структурних одиниць IMDG належить кеш, інша назва якого – регіон (Рис. 1.2). Кеш розподіленого сховища даних в оперативній пам’яті є розподіленим асоціативним масивом. Даний масив, як альтернатива суворо типізованим реляційним базам даних, зберігає серіалізовані об’єкти. Це дозволяє уникнути витрат на десеріалізацію з боку клієнта у процесі зчитування ним даних зі сховища. Зазначена організація уможливлює забезпечення високого рівня горизонтальної масштабованості. І ця обставина є надзвичайно важливою для новітніх веб-сервісів [1].

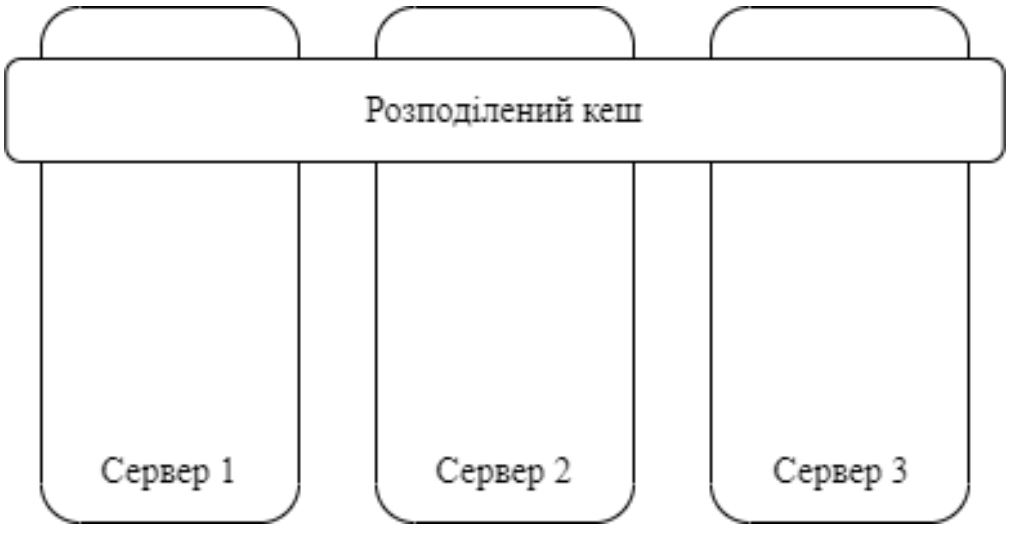


Рисунок 1.2. Організація кешу в розподілених сховищах даних

Головною парадигмою для IMDG систем є BASE – basically available, soft state, eventual consistency, яку розробили виключно для горизонтально масштабованих розподілених систем. Щодо узгодженості, то згідно цієї парадигми, консистентність даних знаходиться у стані потоку. Цей потів, відповідно, постійно змінюється.

Для сховища принциповим залишається забезпечення кінцевої узгодженості даних. Таким чином дані будуть узгоджені за умов відсутності змін. І навіть через певний час після останніх змін. Завдяки таким обставинам, дані не завжди будуть доступні. Коли дані знаходяться в різних частинах сховища і не стали узгодженими, запити до них можуть не повертати результатів [3].

Окремо акцентуємо увагу на підтримці транзакційності IMDG. Завдяки їй задовільняються парадигми ACID. З метою забезпечення механізму узгодженості даних у кластері використовується двофазна фіксація. В різних IMDG відзначаються різні механізми блокування. Однак в найбільш сучасних реалізаціях переважно використовуються паралельні блокування. Цим самим зводиться мережевий обмін до 15 мінімуму та гарантується транзакційна узгодженість ACID зі збереженням високої продуктивності. Різним реалізаціям технології IMDG властиво багато спільних базових функціональних можливостей. Однак одночасно існує багато додаткових можливостей, що мають залежність від конкретного продукту.

Значна увага повинна надаватися механізмам вигрузки даних черех їх переповнення (eviction policies). А також слід контролювати технології завантаження даних, особливо у процесі старту сервера ((pre)loading techniques). Окремої зосередженості потребують паралельне розподілення на секції (concurrent repartitioning) та об’єм додаткової пам’яті, що необхідний для збереження записів (data overhead).

Зберігання даних в IMDG є лише половиною функціоналу, що необхідний для in-memory архітектури. Щодо даних, які зберігають в IMDG, то їх необхідно обробляти одночасно і з високою швидкістю. Як правило, in-memory архітектура секціонує дані в кластері за допомогою IMDG. І вже згодом виконуваний код відправляється на сервери, де знаходиться необхідна йому інформація.

Виконуваний код (обчислювальна задача) зазвичай є частиною обчислювальних кластерів (Computer Grids). Тому він повинен бути правильно розвернутий (deployment). А також збалансованим під час навантаження (load-balancing), бути відмовостійким (fault-tolerant). Окрім зазначеного - мати можливість запуску по розкладці (scheduling). Не менш важлива інтеграція між Compute Grid та IMDG. Вкрай ефективною можна вважати систему, в якій IMDG та Compute Grid є частинами одного і того ж продукту. Вони використовують ідентичні API, що забезпечує можливість досягнути найбільш продуктивного та надійного in-memory рішення.

Охарактеризуємо ряд переваг IMDG перед реляційними БД:

1. **Швидкодія.** Уся інформація зберігається в оперативній пам'яті кластера. За рахунок цього істотно скорочується час доступу.

2. **Відмовостійкість.** На кожному сервері кластеру зберігається копія усього набору даних у сховищі. Таким чином – на випадок виходу з ладу одного чи декількох серверів не буде складно відновити дані, які в даний момент не доступні.

3. **Маштабованість.** Складу кластера (кількість вузлів) властиво змінюватися без зупинки роботи всього кластера. За коректною роботою кластера і 20 консистентним і доступністю даних стежить сам grid без будь-якої допомоги програміста. Отже, не зважаючи на зростаюче навантаження або обсяги даних, можна лише підняти ще кілька сконфігурованих вузлів. Вони автоматично приєднаються до кластера, а дані всередині самого кластера

перебалансуються для рівномірного розподілу даних по вузлах. За таких обставин обсяг переміщених даних буде мінімальний, без створення зайвого навантаження на мережу.

4. **Актуальність даних.** У ході використання IMDG користувач завжди отримує актуальні дані. Адже що при зміні стану даних на всі вузли кластера надсилаються актуальні копії останнього набору даних сховища.

Разом із тим відзначимо, що обчислення в оперативній пам'яті, як і будь-яка технологія, не позбавлені унікальних особливостей, проблем та підводних каменів. По-перше, мова йде про досить велику ціну, адже вона тільки останнім часом зменшується з причини різкого падіння вартості на носії RAM.

Бажано працювати із потужними серверами, багатоядерними процесорами і тоннами оперативної пам'яті. Бажано мати відповідні ПЗ і аналітичні програми. Технологія швидкісної обробки вимагає застосування всіх перерахованих компонентів. Адже терабайти даних зберігаються з «нульовою» латентністю доступу безпосередньо в оперативній пам'яті серверів, а не на дисках.

Ще однією проблемою технології обчислень в оперативній пам'яті є те, що вона добре підходить тільки для транзакцій з наборами структурованих даних. До їх числа належать: таких як артикули товарів, інформація про покупців, звіти з продажу.

У разі, якщо компанії має в своєму розпорядженні засоби та розуміє цінність інформації в сучасній бізнес-стратегії, то технологію обчислень в оперативній може стати для неї підходящим вибором.

1.2 Тестування продуктивності In-Memory Data Grid (IMDG) систем

Відповідно до опублікованих тестів McObject (Рис.1.3), у ході проведення яких порівнювали продуктивність одного і того ж додатку, перенесення традиційної СУБД в RAM дозволило прискорити зчитування даних в 4 разию Також відбулосяа оновлення бази в 3 рази в порівнянні з традиційним СУБД на жорсткому диску. У пам'яті СУБД показала ще більш істотні результати у порівнянні з СУБД на RAM дисках. Зокрема читання баз даних було в 4 рази швидше, а запис в базу даних виявився швидшим в 420 разів.

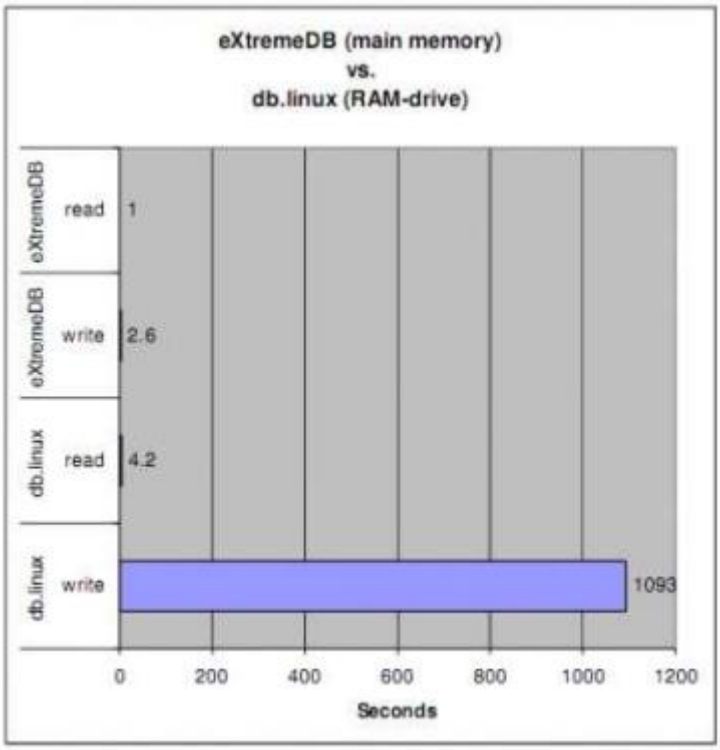


Рисунок 1.3 – продуктивність в пам'яті СУБД eXtremeDB в порівнянні з СУБД db.linux на RAM диску

Головна відмінність in-memory СУБД від традиційних СУБД є те, що in-memory СУБД не має у своїй собіструктурова навантаження від операцій введення / виводу даних. Тому архітектура таких баз даних більш є раціональнас Процеси завантаження пам'яті і цикли процесора оптимізовані.

In-memory СУБД зазвичай використовуються для додатків, які вимагають надшвидкого доступу до даних, сховищ і маніпуляцій з ними, а ткож в системах, які не мають жорсткого диску, але, тим не менш, повинні управляти значною кількістю даних. Згідно зі звітом McObject, in-memory СУБД легко можна масштабувати за розмірами, що перевищують терабайт.

Таким чином, у ході проведених тестів 64-бітна inmemory СУБД встановленій на 160-ядерному сервері SGI Altix 4700 під керуванням SUSE Linux Enterprise Server версії 9 від Novell вдалося досягнути 1,17 терабайт і 15,54 млрд рядків без видимих обмежень для подальшого масштабування. Що ж до продуктивності в даному тесті, то вона практично не змінювалася. Навіть в міру того, як СУБД досягла сотень гігабайт, а потім і терабайта. І цей факт свідчить про практично лінійну масштабованість.

1.3 Порівняльна характеристика найпопулярніших IMDG систем

В таблиці 1.1 показані основні характеристики існуючих IMDG систем.

Таблиця 1.1. – Порівняльна характеристика найпопулярніших IMDG продуктів

Name	**GridGain**	**Hazelcast IMDG**	**NCache**	**Oracle Coherence**	**Redis**
Secondary database models	no	Document store;	Document store; Search engine;	no	Document store; Graph DBMS; Search engine; Time series DBMS;

Продовження таблиці 1.1. – Порівняльна характеристика найпопулярніших IMDG продуктів					
Current release	GridGain 8.5.1	4.0, February 2020	5.0, May 2019	12c, September 2016	5.0.9, April 2020
Primary database model	Key-value store; Relational DBMS;	Key-value store;	Key-value store;	Key-value store;	Key-value store;
Cloud-based only	no	no	no	no	no
Implementation language	Java; C++; .NET;	Java	C#; .NET; .NET Core;	Java	C
Server operating systems	Linux; OSX; Solaris; Windows;	All OS with a Java VM	Linux; Windows;	All OS with a Java VM	BSD; Linux; OS X; Windows;
Typing	yes	yes	partial (Lists, Queues, Hashsets, Dictionary and Counter)	yes	partial (strings, hashes, lists, sets and sorted sets, bit arrays, hyperloglog)
Data scheme	yes	schema-free	schema-free	schema-free	schema-free

Продовження таблиці 1.1. – Порівняльна характеристика найпопулярніших IMDG продуктів					
XML support	yes	yes	no	no	no
SQL	ANSI-99 for query and DML statement	SQL-like query language	SQL-like query syntax and LINQ	no	no
APIs and other access methods	HDFS API; Hibernate; JCache; JDBC; ODBC; Proprietary protocol; RESTful; HTTP API; Spring; Data;	JCache; JPA; Memcached protocol; RESTful; HTTP API;	IDistributedCache JCache; LINQ; Proprietary native API;	JCache; JPA; RESTful HTTP API;	proprietary protocol RESP;
Server-side scripts	yes (compute grid and)	yes (Event Listeners, Executor Services)	no	no	yes (Lua)
Triggers	yes (cache interceptors and events)	yes (Events)	yes (Notifications)	yes (Live Events)	no
Partitioning methods	Sharding	Sharding	Sharding	Sharding	Sharding

Продовження таблиці 1.1. – Порівняльна характеристика найпопулярніших IMDG продуктів

Replication methods	yes (replicated cache)	yes (Replicated Map)	yes, with selectable consistency level	yes, with selectable consistency level	yes (Master-slave replication; Multi-master replication;)
MapReduce	yes	yes	yes	no	no
Transaction concepts	ACID	one or two-phase-commit; repeatable reads; read commited;	optimistic locking and pessimistic locking	configurable	Optimistic locking; atomic execution of commands blocks and scripts;
Concurrency	yes	yes	yes	yes	yes
Durability	yes	yes	yes	yes	yes
In-memory capabilities	yes	yes	yes	yes	yes
Access control	Security Hooks for custom implementations	Role-based access control	Authentication to access the cache via Active Directory/LDAP (possible roles: user, administrator)	Authentication to access the cache via certificates	Simple password-based access control

Виходячи з таблиці 1.1, можемо побачити, що всі порівнювані системи реалізують реплікацію для збереження копій усього набору даних. Проте, неможливо точно сказати який з продуктів є кращим у плані відмовостійкості, оскільки частина представлених рішень мають відкриті та платні версії. Безкоштовні версії мають обмежені можливості (локальний IMGD), а платні версії виключають можливість ознайомлення з внутрішньою будовою цих систем. Така ж сама ситуація у питанні балансування навантаження серверів кластеру.

Також, всі порівнювані системи реалізують основний функціонал, який повинні бути притаманний IMDG системі для ефективної взаємодії з нею та отримання оптимальних показників швидкодії.

ВИСНОВОК ДО РОЗДІЛУ 1

В даному розділі було проведено теоретичний аналіз IMDG реалізації розподілених сховищ даних. Розглянуто перелік переваг та недоліків таких систем. Представлено результати тестування продуктивості IMDG системи у порівнянні з класичною базою даних. Проведено порівняльну характеристику найпопулярніших IMDG систем.

Проаналізувавши функціонал систем, наведених у Табл. 1, стало зрозуміло, що наразі існуючі рішення мають або платний доступ, або обмежений функціонал, або недостатньо детально описану документацію щодо моделі та структури системи.

Тому було вирішено розробити IMDG систему з відкримим доступом, яка б відповідала вимогам відмовостійкості та масштабування, підтримувала зберігання та обробку складних структур даних, реалізувала можливість одночасної взаємодії з багатьма користувачами та інше.

РОЗДІЛ 2
ОГЛЯД АРХІТЕКТУРИ РОЗРОБЛЕНОЇ СИСТЕМИ

Архітектура системи описує основні компоненти, загальну структуру, програмне та технічне забезпечення, принципи правильної роботи та взаємодії користувача з системою [10].

Структура розробленої системи складається з трьох основних рівнів: рівень клієнта, рівень планувальників, рівень сховища (Рис.2.1).

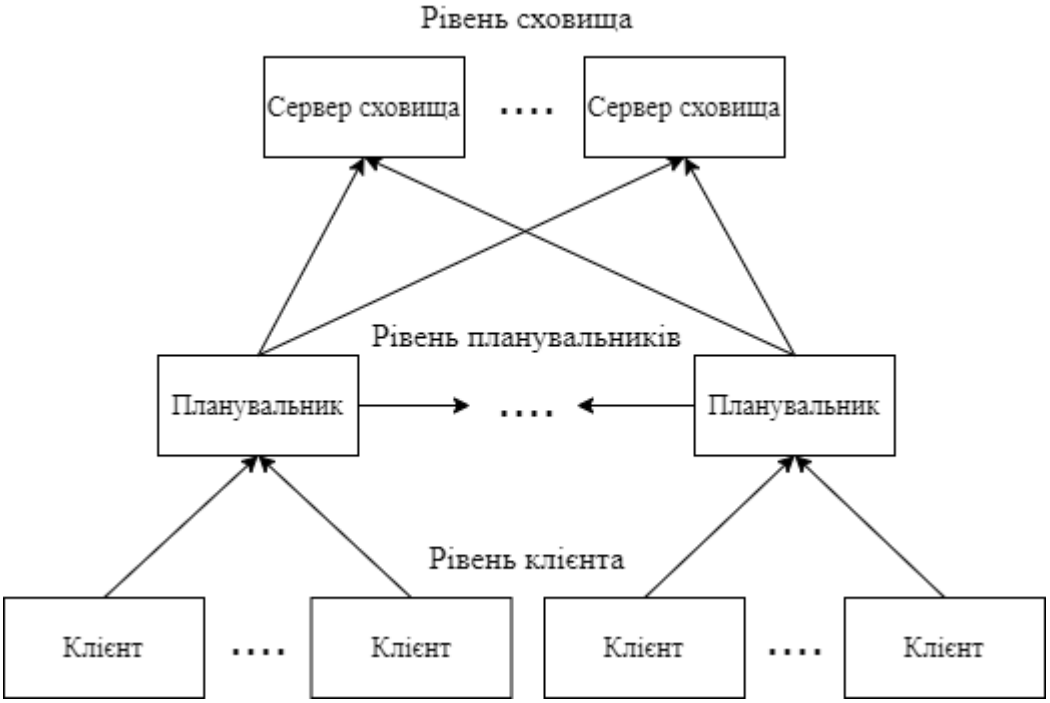


Рисунок 2.1. Загальна структурна схема

Рівень клієнта призначений для надання користувачу інтерфейсу, за допомогою якого він може взаємодіяти з системою.

Рівень планувальників є проміжним між клієнтом та кластером. Сервери цього рівня приймають запити від клієнта і передають їх до рівня сховища. Основними завданнями планувальників є збереження останнього стану даних в кластері (використовуючи постійну пам’ять), постійний моніторинг кластера на наявність змін у сховищі, балансування сховища і синхронізації між усіма серверами цього рівня та всіма серверами кластеру.

Рівень сховища являє собою IMDG кластер, що приймає клієнтські запити від планувальників і виконує їх. Також кожен сервер цього рівня зберігає копію актуального стану даних у сховищі, використовуючи для цього постійну пам’ять.

2.1 Огляд структурних елементів системи

Система реалізує клієнт-серверну архітектуру.

Архітектура клієнт-сервер – це підхід до проектування інфомаційної мережі, при якому основна частина ресурсів розміщується на серверах, які обслуговують клієнтів. У такій архутектурі визначаються наступні типи компонентів – клієнти і сервери.

Сервер – це компомент мережі, що виконує запити інших компонентів, надаючи їм сервіси. Сервіс - це надання певних послуг клієнтам.

Сервер виконує завдання, що надходять від клієнтів. Після завершення обробки сервер надсилає клієнту результат виконання завдання.

Сервісна функція в архітектурі клієнт-сервер забезпечується набором програмного забезпечення, що може виконувати різноманітні завдання.

Клієнт – процес, який викликає сервісну функцію і надає користувачеві інтерфейс створення запитів. Користувацький інтерфейс – це функціонал, що дозволяє користувачу взаємодіяти з системою.

Клієнт ініціює завдання, використовуючи доступні сервіси сервера. Під час цього процесу клієнт встановлює з’єднання, формує запит, отримує результати та інформує сервер про закінчення роботи.

Клієнти та сервери є незалежними один від одного. Часто зустрічаються ситуації, коли один сервер обробляє запити різних клієнтів. З іншого боку, у клієнтів є можливість послідовно звертатися до різних серверів. Клієнти мають інформацію про сервери, до яких вони можуть звертатися. Але, у свою чергу,

вони не повинні володіти інформацією про інших клієнтів, що також взаємодіють з серверами у даний момент часу [8].

Процеси клієнта і сервера знаходяться на різних комп'ютерах, підключених до мережі, хоча можуть перебувати і на одному і тему ж комп'ютері. Перебуваючи в мережі, сервер може надавати сервіси більш ніж одному клієнту, а клієнт може запитувати сервіси від декількох серверів мережі незалежно від їх розташування або фізичних характеристик комп'ютера, на якому знаходиться процес сервера. Мережа надає клієнтам і серверам засоби зв'язку [9].

У даній роботі було вирішено використовувати клієнт-серверну архітектуру, тому що такий підхід є найбільш оптимальним варіантом реалізації поставленої задачі. Наведений архітектурний шаблон дозволяє реалізувати багатокористувацький режим роботи, коли один сервер здатен працювати з багатьма клієнтами одночасно, без особливих затрат часу. Даний підхід надає можливість зменшувати навантаження на робочі станції користувачів системи.

Процеси клієнта і сервера незалежні один від одного. Відповідно до ступеня поділу процесів між клієнтом і сервером вони вважаються слабкими (тонкими) або сильними (товстими). Слабкий клієнт виконує мінімум обробки на стороні клієнта, сильний бере на себе відносно велику частину обробки даних. Сильний сервер несе основне навантаження по обробці даних, навантаження на слабкий сервер відносно невелике.

Клієнт-серверні системи поділяються на двох- і трьохрівневі. У першому випадку клієнт запитує сервіси безпосередньо у сервера, у другому запити обробляються проміжними серверами, які координують виконання клієнтських запитів з підлеглими їм серверами. Крім того, проміжні сервери можуть реалізовувати додатковий функціонал, наприклад реплікацію даних в сховищі [9].

Під час розробки було обрано трьохрівневу модель клієнт-серверної архітектури, оскільки вона дозволяє реалізувати масштабованість системи та реплікацію даних без переміщення даних безпосереднь між серверами кластеру.

Дана система реалізує модель слабкого клієнта та сильного сервера. Вся логіка програми зосереджена на сервері, який представлений набором серверів-планувальників та розподіленим сховищем даних в оперативній пам’яті.

2.1.1 Клієнтська частина

Клієнтська частина призначена для надання користувачу інтерфейсу, за допомогою якого він може взаємодіяти з системою. Взаємодія відбувається за допомогою механізмів клієнтських запитів та серверних відповідей. Клієнтська частина реалізує можливість клієнта формувати запити, відправлення запитів до серверної частини, відображення отримах результатів.

Для реалізації клієнтської частини системи було обрано мову програмування C# та середовище розробки програмного забезпечення MS Visual Studio 2019 Community Edition.

C# – строго типізована об'єктно-орієнтована мова програмування, призначена для розробки різноманітних безпечних і потужних додатків, які виконуються в середовищі .NET Core. За допомогою C# можна розробляти звичайні клієнтські програми Windows, веб-служби XML, розподілені компоненти, додатки типу клієнт-сервер, додатки баз даних.

.NET Core – середовище розробки загального призначення з відкритим кодом, призначене для створення кроссплатформенних додатків. Ви можете створювати додатки .NET Core для Windows, macOS і Linux з підтримкою процесорів x64, x86, ARM32 і ARM64, використовуючи кілька мов програмування.

Основні переваги середовища розробки .NET Core:

- Підтримка операційних систем Windows, macOS, Linux;
- Платформа .NET базується на відкритому коді;
- Підтримка сучасних технологій, таких як асинхронне програмування, шаблони без копіювання з використанням структур та керування ресурсами для контейнерів;
- Забезпечує високу продуктивність роботи програм;
- Забезпечує ідентичне виконання коду в різних операційних системах;
- Зручні засоби локальної розробки;

Microsoft Visual Studio – інтегроване середовище розробки (IDE), що являє собою багатофункціональну програму, яку можна використовувати для різних аспектів розробки програмного забезпечення. Крім стандартного редактора і відладника, які існують в більшості середовищ, Microsoft Visual Studio включає в себе компілятори, засоби автозавершения коду, графічні конструктори і багато інших функцій для спрощення процесу розробки.

Основні переваги середовища розробки MS Visual Studio 2019 Community Edition:

- Зручний редактор та користувацький інтерфейс;
- Постійна підтримка продукту розробниками;
- Інтерактивне відображення помилок у коді;
- Інтерактивні підказки альтернативної реалізації сумнівних ділянок коду;
- Інтерактивні підказки з описом методів та їх параметрів;
- Функція автостворення методів, які обов’язково повинні бути реалізовані у програмі;
- Висока популярність серед розробників, що дозволяє знайти готове рішення більшості проблем, пов’язаних з середовищем розробки;

– Велика кількість інтегрованих бібліотек;
– Потужний інструментарій для відладки програм, що дозволяє досить швидко знаходити та виправляти помилки;
– Інтегровані засоби тестування програм;
– Інтегровані засоби мережевої розробки;

Під час проектування архітектури системи було обрано саме це програмне середовище через простоту у налаштуванні та наявність всього необхідного функціоналу для забезпечення зв’язків зі сховищем даних, багатокористувацького режиму роботи програмного забезпечення та простоти інтеграції системи на інші операційні системи при виникненні такої необхідності.

2.1.2 Серверна частина

Серверна частина призначена для обробки отриманих від клієнта запитів та відправлення результатів обробки цих запитів клієнтському додатку. У свою чергу, серверна частина складається з двох незалежних рівнів: рівня планувальників та рівня сховища даних в оперативній пам’яті.

2.1.2.1 Рівень планувальників

Сервери рівня планувальників приймають запити від клієнта і передають їх до рівня сховища. Крім того, цей рівень необхідний для організації відмовостійкості та маштабованості системи.

У аспекті відмовостйкості, головними завданнями планувальників є збереження останнього стану даних у кластері, постійний моніторинг кластера на наявність змін стану сховища, синхронізація актуальних даних з усіма серверами рівня планувальників та рівня сховища.

Зі сторони клієнта, взаємодія з системою виглядає так, наче клієнт напряму під’єднується до сховища і починає взаємодіяти з ним. Насправді ж,

клієнт під’єднується до одного з планувальників, що обирається в залежності від завантаженості системи. Всі клієнтські запити надходять до планувальника, який у свою чергу надсилає їх до потрібного сервера сховища. Далі сервер сховища виконує запит і надсилає відповідь планувальнику, а той клієнту.

Кожен планувальник може одночасно обслуговувати більше одного клієнта. Це дозволяє системі у один момент часу асинхронно виконувати величезну кількість запитів від багатох клієнтів.

Для забезпечення відмовостійкості системи були розроблені алгоритми збереження копій останнього стану даних у сховищі. Ключову роль у цих алгоритмах відіграють саме сервери рівня планувальників.

Рис. 2.2 демонструє як саме система створює та розсилає копію всього набору даних відповідним її компонентам при оновленні стану даних.

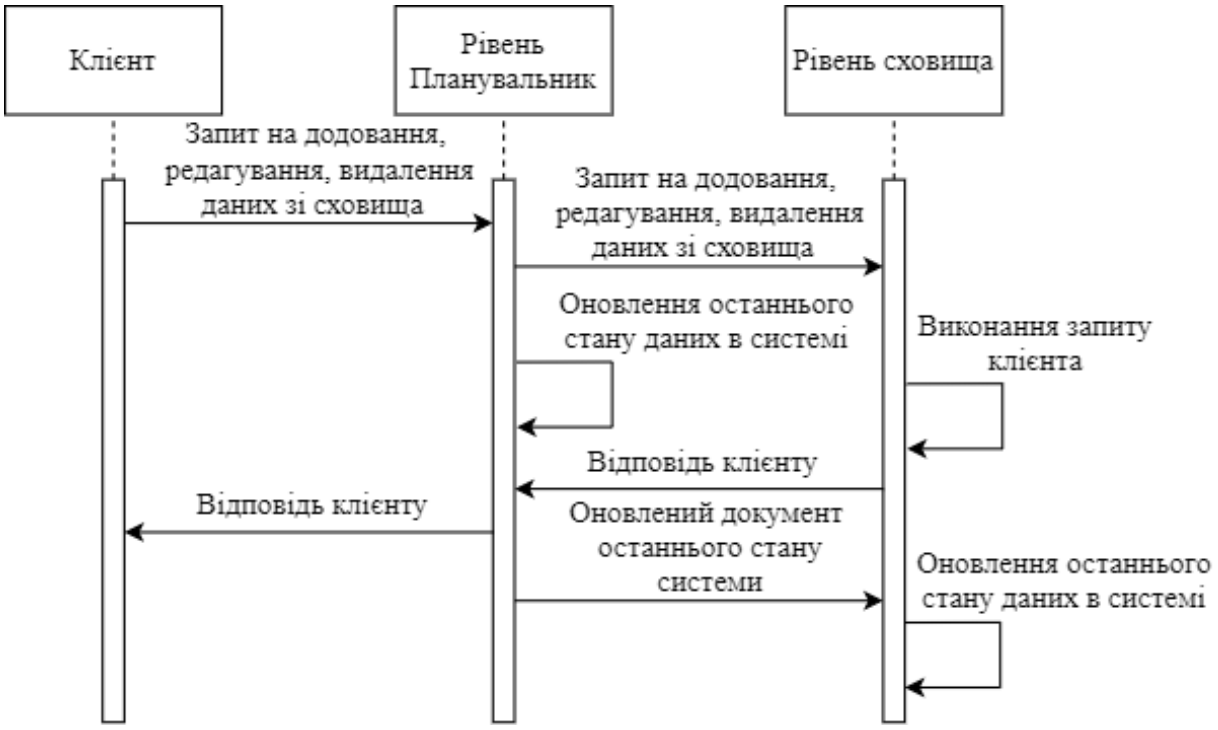


Рисунок 2.2. Алгоритм дій системи при додаванні, редагуванні, видаленні даних з кластера

Розглянемо етапи алгоритму, зображеного на Рис. 2.2:

1. Планувальник отримує запит від клієнта і, відповідно до механізму балансування, передає запит на потрібний сервер кластеру;
2. Планувальник оновлює свій файл останнього стану кластера;
3. Планувальник відправляє оновлений файл останнього стану системи всім іншим серверам рівня планувальників та всім серверам рівня сховища;
4. Всі сервери, які отримали файл, оновлюють власні файли останнього стану системи;
5. Сервер рівня сховища, який виконав запит клієнта, надсилає відповідь планувальнику;
6. Планувальник надсилає відповідь клієнту;

Рис. 2.3 демонструє як саме система може відновити дані, якщо виходить з ладу один з серверів кластеру. Використовуючи копію всього набору даних, система визначає що саме знаходилося на недоступному сервері, здійснює балансування системи з урахуванням нової кількості доступних серверів та завантажує дані в їхню оперативну пам’ять.

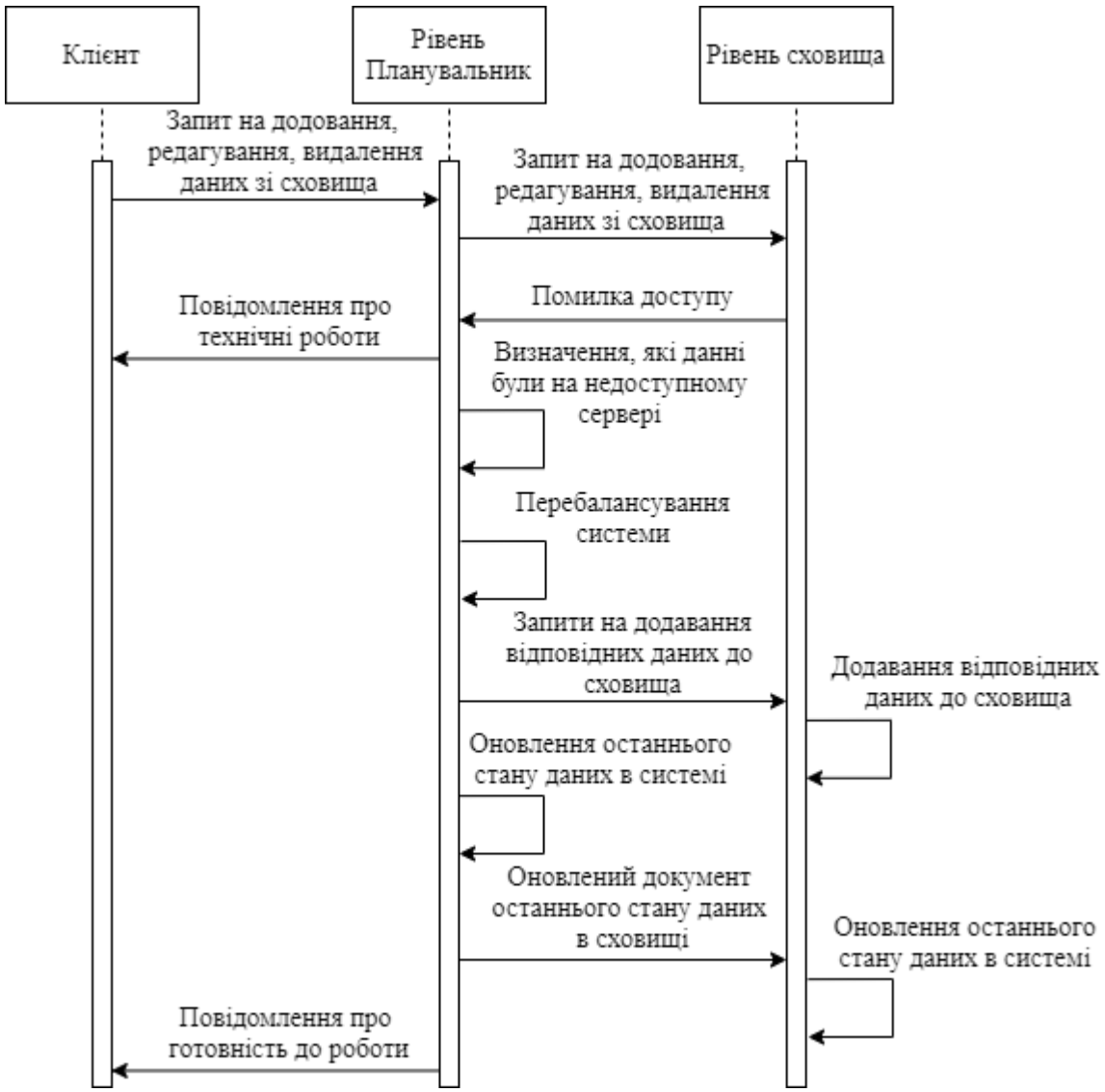


Рисунок 2.3. Алгоритм дій системи у разі виходу з ладу одного з серверів кластерного рівня

Розглянемо етапи алгоритму, зображеного на Рис. 2.3:

1. Планувальник фіксує, що один з серверів кластеру не доступний;
2. Планувальник надсилає клієнтам повідомлення про технічні роботи;
3. Планувальник визначає, які дані були на сервері, що наразі є недоступним;
4. Відповідно до механізму балансування, відбувається розрахунок того, які сервери кластера повинні завантажити до своєї оперативної пам’яті дані недоступного сервера кластера;

5. Планувальник відправляє запити на додавання даних відповідним серверам сховища;

6. Сервери рівня сховища додають до оперативної пам’яті необхідні дані;

7. Планувальник оновлює свій файл останнього стану даних;

8. Планувальник відправляє оновлений файл останнього стану системи всім іншим планувальникам та всім серверам рівня сховища;

9. Всі сервери, які отримали файл, оновлюють власні файли останнього стану системи;

10. Планувальник надсилає клієнту повідомлення про готовність до роботи;

Рис. 2.4 демонструє як саме система може відновити свій попередній стан після запланованого або незапланованого вимкнення усіх серверів кластеру. Такий підхід дозволяє проводити технічне обслуговування всього кластеру без ризику втрати завантажених в кеш даних.

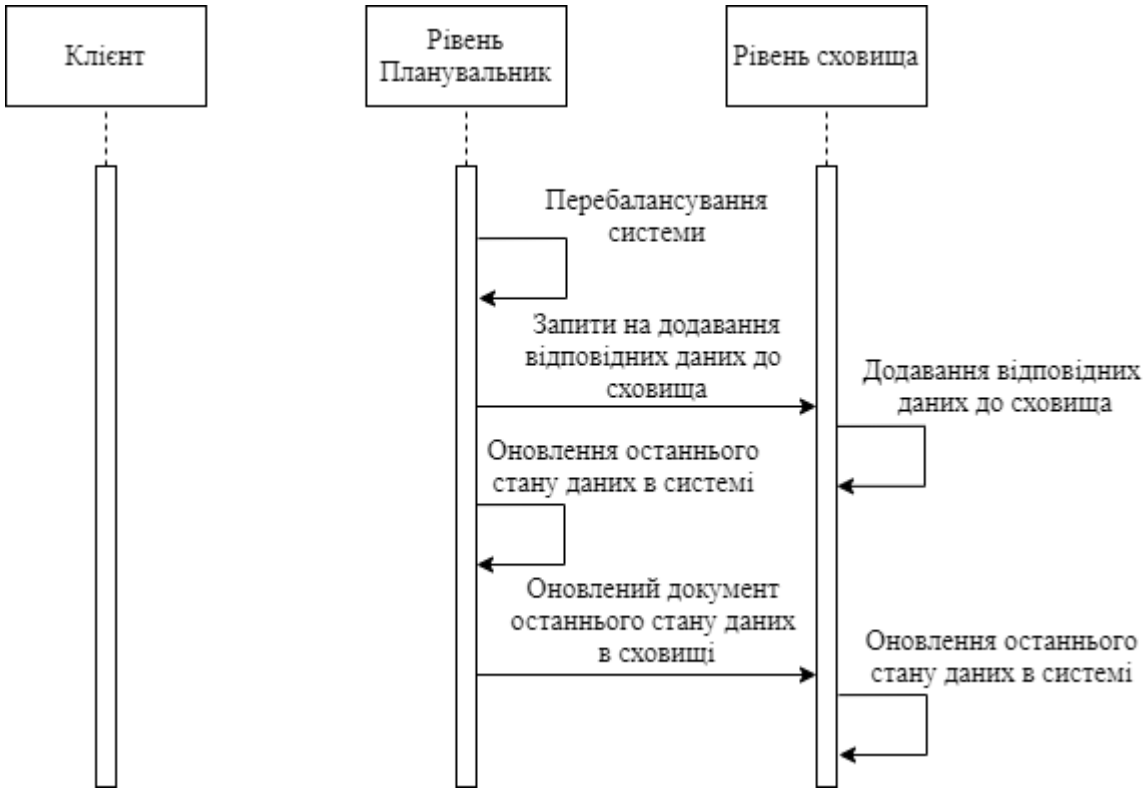


Рисунок 2.4. Алгоритм відновлення стану системи після перезавантаження кластеру

Розглянемо етапи алгоритму, зображеного на Рис. 2.4:

1. Сервер рівня Master Level визначає, які дані були на кожному сервері кластеру;
2. Сервер рівня Master Level відправляє запити на додавання даних відповідним серверам;
3. Сервери кластерного рівня додають до своєї оперативної пам’яті необхідні дані;
4. Сервер рівня Master Level оновлює свій файл останнього стану кластера;
5. Сервер рівня Master Level відправляє оновлений файл останнього стану системи всім іншим серверам рівня Master Level та всім серверам кластерного рівня;
6. Всі сервери, які отримали файл, оновлюють власні файли останнього стану системи;

Зважаючи на переваги, описані в пункті 2.1.1, для реалізації рівня планувальників було обрано мову програмування C# та середовище розробки програмного забезпечення MS Visual Studio 2019 Community Edition.

2.1.2.2 Рівень сховища даних в оперативній пам’яті

Даний рівень являє собою IMDG кластер. Сервери цього рівня приймають клієнтські запити від планувальників і виконують їх. Також кожен сервер цього рівня зберігає копію останнього стану даних у сховищі, використовуючи для цього постійну пам’ять.

Хеш-таблиця – структура даних, що дозволяє зберігати дані у вигляді пари ключ-значення.

Сервери цього рівня використовують свою оперативну пам’ять для зберігання даних. По суті, на кожному такому сервері створюється хеш-таблиця, розмір якої обмежений вмістимістю оперативного запам’ятовуючого

пристрою. З точки зору клієнта, такий кластер сприймається як одна велика хеш-таблиця, у якій і зберігаються дані.

Основною вимогою до сховища є можливість зберігання та обробка складних структур даних, таких як хеш-таблиця або список. Крім того, клієнт повинен мати можливість збереження даних у вигляді простого рядка.

Зважаючи на переваги, описані в пункті 2.1.1, для реалізації рівня планувальників було обрано мову програмування C# та середовище розробки програмного забезпечення MS Visual Studio 2019 Community Edition.

ВИСНОВОК ДО РОЗДІЛУ 2

В даному розділі описані найбільш оптимальні архітектурні рішенння для побудови як клієнтської, так і серверної частин системи.

Основною перевагою представленої моделі системи є відсутність переміщень даних всередині кластера. Якби дані переміщувалися безпосередньо між серверами кластеру, то це призвело б до необхідності перебалансовувати кластер після кожного такого переміщення. А це в свою чергу призводить до суттєвого зниження швидкодії усієї системи.

Середовище розробки .NET Core було обрано з огляду на підтримку сучасних технологій програмування, високу продуктивність програм, інструменти мережевої розробки.

Інтегроване середовище розробки Microsoft Visual Studio було обране з огляду на зручний користувацький інтерфейс, вбудовані інструменти відладки та тестування коду програми.

РОЗДІЛ 3
ОПИС ПРОГРАМНОЇ РЕАЛІЗАЦІЇ

3.1 Клієнтська частина

Клієнтський додаток реалізований у середовищі розробки MS Visual Studio мовою програмування C#. Інтерфейс користувача представлений у вигляді консольної програми. Вся логіка системи реалізована в серверній частині, тому завданням клієнтського додатку є лише надсилання запитів до серверної частини та відображення отриманих результатів.

3.1.1 Перелік модулів комп'ютерної програми

Програмний код проекту містить 1 модуль:

– Client: головний модуль клієнтської програми, описує клас Client;

3.1.2 Користувацькі класи та методи

Для відправки запитів, отримання та відображення відповідей серверної частини розроблено користувацькі класи та методи цих класів, специфікацію яких наведено нижче.

Клас Client – головний клас програми, відповідає за підключення до серверної частини, відправку запитів та отримання відповідей.

Методи класу:

– static async Task Main(string[] args) – точка входу в програму;

– private static async Task SendRequest(Stream stream, string request, Memory<byte> byteWrite) – асинхронний метод надсилання клієнтських запитів до серверної частини;

– private static async Task<string> GetAnswer(Stream stream, Memory<byte> byteRead) – асинхронний метод отримання відповіді серверної частини;

3.1.3 Підключення до серверної частини

Під час запуску користувацького додатку, програма автоматично під’єднується до одного з серверів рівня планувальників. Підключення здійснюється за допомогою класу TcpClient із пакета System.Net.Sockets. У разі успішного підключення, на консоль виводиться повідомлення про готовність до роботи. У випадку помилки з’єднання, на консоль виводиться відповідне повідомлення.

3.1.4 Додавання, видалення, редагування, отримання даних зі сховища

Зі сторони клієнта, алгоритми додавання, видалення, модифікації та отримання даних зі сховища являють собою введення відповідного запиту до консолі і отримання результату від серверної частини. Вся логіка, така як перевірка правильності запиту, взаємодія зі сховищем, балансування сховища та реплікація даних реалізована у сервеній частині системи.

3.2 Серверна частина

Серверна частина реалізована у середовищі розробки MS Visual Studio мовою програмування C#. У даній частині зосереджена вся логіка роботи системи.

3.2.1 Перелік модулів комп’ютерної програми

Програмний код проекту містить 17 модулів:

– Command: модуль, що описує класи усіх доступних команд;

– CommandKind: модуль, що містить перечислення усіх ідентифікаторів доступних команд;

– CommandFacts: модуль, що визначає які типи команд будуть виконуватися під час балансування системи;

– FileHelper: модуль, що відповідає за читання усього набору даних з файлу;

– PartitionRange: модуль, що описує клас для визначення діапазонів значень для кожного сервера кластера;

– StringExtentions: модуль, що відповідає за обчислення унікального ідентифікатора для кожного типу команд в системі;

– Value: модуль, що описує класи структур даних;

– ClientServer: модуль, що відповідає за керування асинхронними з'єднаннямии з клієнтами;

– FileWritter: модуль, що відповідає за запис усього набору даних у файл;

– Manager: модуль, що відповідає за передачу даних від клієнта до сховища, моніторингу сховища на необхідність балансування;

– ManagerMain: головний модуль рівня планувальників, що відповідає за запуск планувальника;

– StorageServer: модуль, що відповідає за керування асинхронними з'єднаннямии зі сховищами;

– TcpServerBase: модуль, що описує абстрактний клас для керування асинхронними з'єднаннями;

– Parser: модуль, що описує алгорим перевірки правильності користувацького запиту;

– FileHandler: модуль, що описує роботу серверів кластерного рівня з іхніми копіями усього набору даних;

– Storage: модуль, що описує сервер рівня сховища;

– StorageMain: головний модуль рівня сховища, що відповідає за запуск сервера кластера;

3.2.2 Користувацькі класи та методи

Для реалізаії відправки запитів, отримання та відображення відповідей серверної частини створено користувацькі класи та методи цих класів, специфікацію яких наведено нижче.

Клас Command – абстрактний клас, що описує запит без параметрів. Не реалізує методів.

Клас KeyCommand – абстрактний клас, що описує запит з одним параметром. Не реалізує методів.

Клас KeyValueCommand – абстрактний клас, що описує запит з двома параметрами. Не реалізує методів.

Клас GetCommand – клас, що описує команду get.

Методи класу:

– public override CommandKind Kind => CommandKind.Get – метод, повертає команду типу get;

Клас SetCommand – клас, що описує команду set.

Методи класу:

– public override CommandKind Kind => CommandKind.Set – метод, повертає команду типу set;

Клас GetAllCommand – клас, що описує команду getall.

Методи класу:

– public override CommandKind Kind => CommandKind.GetAll – метод, повертає команду типу getall;

Клас UpdateCommand – клас, що описує команду update.

Методи класу:

– public override CommandKind Kind => CommandKind.Update – метод, повертає команду типу update;

Клас GetKeysCommand – клас, що описує команду getkeys.

Методи класу:

– public override CommandKind Kind => CommandKind.GetKeys – метод, повертає команду типу getkeys;

Клас RemoveCommand – клас, що описує команду remove.

Методи класу:

– public override CommandKind Kind => CommandKind.Remove – метод, повертає команду типу remove;

Клас FindCommand – клас, що описує команду find.

Методи класу:

– public override CommandKind Kind => CommandKind.Find – метод, повертає команду типу find;

Клас ClearCommand – клас, що описує команду clear.

Методи класу:

– public override CommandKind Kind => CommandKind.Clear – метод, повертає команду типу clear;

Клас HSetCommand – клас, що описує команду hset.

Методи класу:

– public override CommandKind Kind => CommandKind.Hset – метод, повертає команду типу hset;

Клас HGetAllCommand – клас, що описує команду hgetall.

Методи класу:

– public override CommandKind Kind => CommandKind.HGetAll – метод, повертає команду типу hgetall;

Клас HRemoveCommand – клас, що описує команду hremove.

Методи класу:

– public override CommandKind Kind => CommandKind.HRemove – метод, повертає команду типу hremove;

Клас HGetCommand – клас, що описує команду hget.

Методи класу:

– public override CommandKind Kind => CommandKind.HGet – метод, повертає команду типу hget;

Клас HValCommand – клас, що описує команду hval.

Методи класу:

– public override CommandKind Kind => CommandKind.HVal – метод, повертає команду типу hval;

Клас HKeysCommand – клас, що описує команду hkeys.

Методи класу:

– public override CommandKind Kind => CommandKind.HKeys – метод, повертає команду типу hkeys;

Клас LAddCommand – клас, що описує команду ladd.

Методи класу:

– public override CommandKind Kind => CommandKind.LAdd – метод, повертає команду типу ladd;

Клас LGetAllCommand – клас, що описує команду lgetall.

Методи класу:

– public override CommandKind Kind => CommandKind.LGetAll – метод, повертає команду типу lgetall;

Клас LRemoveCommand – клас, що описує команду lremove.

Методи класу:

– public override CommandKind Kind => CommandKind.LRemove – метод, повертає команду типу lremove;

Клас LCountCommand – клас, що описує команду lcount.

Методи класу:

– public override CommandKind Kind => CommandKind.LCount – метод, повертає команду типу lcount;

Клас CommandFacts – клас, що визначає які типи команд будуть виконуватися під час балансування системи;.

Методи класу:

– public static bool CanReplay(Command command) – метод, що містить типи команд, які будуть виконуватися під час балансування системи;

Клас FileHelper – клас, що відповідає за читання усього набору даних з файлу;

Методи класу:

– public static async Task<IEnumerable<Command>> ReadCommandsFromFile (string fileName, Encoding encoding = null) – метод, що відповідає за асинхронне читання даних з файлу усього набору даних;

– public static async Task TruncateAsync (string fileName, IEnumerable<Command> commands, Encoding encoding = null) – метод, що відповідає за перезапис даних до файлу останнього стану системи;

Клас PartitionRange – клас, що відповідає за визначення діапазонів для кожного серверу кластера;

Методи класу:

– public static PartitionRange All() – метод, що повертає повний діапазон значень;

– public bool Equals(PartitionRange other) – метод, що порінює діапазони значень;

– public override int GetHashCode() – метод, що відповідає за отримання хеш-коду від команд;

– public bool Contains(int value) – метод, що визначає чи належить число відповідному діапазону;

Клас StringExtentions – клас, що відповідає за обчислення унікальних ідентифікаторів для кожного типу команд в системі;

Методи класу:

– public static int GetDeterministicHashCode(this string str) – метод, що визначає унікальних ідентифікатор для кожного типу команд в системі;

Клас StringExtentions – клас, що відповідає за обчислення унікальних ідентифікаторів для кожного типу команд в системі;

Методи класу:

– public static int GetDeterministicHashCode(this string str) – метод, що визначає унікальних ідентифікатор для кожного типу команд в системі;

Клас Value – клас, що реалізує абстракцію даних.

Методи класу:

– public abstract string Display() – метод, що відповідає за відображення даних;

Клас StringValue – клас, що реалізує роботу з даними типу рядок.

Методи класу:

– public override string Display() – метод, що відповідає за відображення даних типу рядок;

Клас HashValue – клас, що реалізує роботу з даними типу хеш-таблиця.

Методи класу:

– public override string Display() – метод, що відповідає за відображення даних типу хеш-таблиця;

– public string GetValues() – метод, що відповідає за відображення усіх значень у хеш-таблиці;

– public string GetKeys() – метод, що відповідає за відображення усіх ключів у хеш-таблиці;

Клас ListValue – клас, що реалізує роботу з даними типу список.

Методи класу:

– public override string Display() – метод, що відповідає за відображення даних типу список;

Клас ClientServer – клас, що відповідає за керування асинхронними з’єднаннямии з клієнтами;

Методи класу:

– protected override async Task HandleConnectionAsync(TcpClient tcpClient, Guid _) – метод, що відповідає за обмін повідомленнями між серверною частиною та клієнтом;

– private static async Task<string> GetRequest(Stream stream, Memory<byte> byteRead) – метод, що відповідає за отримання запиту від клієнта;

– private static async Task SendAnswer(Stream stream, string answer, Memory<byte> byteWrite) – метод, що відповідає за відправку відповіді клієнту;

Клас FileWriter – клас, що відповідає за запис усього набору даних у файл;

Методи класу:

– public Task ReadMessages() – метод, що відповідає за запис даних у файл;

– public async void FileSync() – метод, що відповідає за відправку файлу останнього стану системи до серверів кластерного рівня;

Клас Manager – клас, що відповідає за запуск і роботу планувальника;

Методи класу:

– private static IList<PartitionRange> NewRanges(int count) – метод, що відповідає за перерахунок діапазонів значень;

– public async Task HandleStorage(TcpClient client, Guid id) – метод, що відповідає за початок балансування сховища;

– private async Task DoRebalance(TcpClient client, Guid id) – метод, що відповідає за виконання балансування сховища;

– public async Task<string> HandleRequest(string req) – метод, що відповідає за роботу з клієнтськими запитами;

– private async Task<string> SendRequest(Command req, Storage client) – метод, що відповідає за надсилання клієнтських запитів до сховища з урахуваннями необхідності балансування;

– public async Task SendSync(IEnumerable<Command> commands) – метод, що відповідає за надсилання сигналу про необхідність перебаласування сховища;

Клас ManagerMain – клас, що відповідає за запуск планувальника;

Методи класу:

– private static async Task Main() – метод, що відповідає за запуск планувальника;

Клас StorageServer – клас, що відповідає за керування асинхронними з’єднаннямии з серверами сховища;

Методи класу:

– protected override Task HandleConnectionAsync(TcpClient tcpClient, Guid clientId) – метод, що відповідає за обмін повідомленнями між планувальником та сервером сховища;

Клас TcpServerBase – абстрактний клас для керування асинхронними з’єднаннями;

Методи класу:

– public async Task Start() – метод, що відповідає за створення з’єднання між клієнтом та сервером;

– private async Task StartHandleConnectionAsync(TcpClient tcpClient, Guid clientId) – метод, що відповідає за можливість серверу асинхронно працювати з багатьма клієнтами;

Клас Parser – клас, що відповідає за перевірку отриманого запиту на валідність та визначення типу команди.

Методи класу:

– public Command Parse() – метод, що відповідає за перевірку отриманого запиту на валідність та визначення типу команди;

– private Command ParseCommand() – метод, що відповідає за визначення типу команди;

– private string MatchWord() – метод, що відповідає за зчитування послідовності символів запиту до знаку пробілу;

– private void MatchEof() – метод, що відповідє за визначення кінця запиту;

– private void SkipWhitespace() – метод, що відповідає за переформатування запиту у послідовність символів без пробілів;

Клас FileHandler – клас, що описує роботу серверів кластерного рівня з іхніми копіями усього набору даних;

Методи класу:

– public Task UpdateFileAsync(IEnumerable<Command> commands) – метод, що відповідає за оновлення файлу останнього стану системи серверів кластерного рівня;

– private static string GetFileName() – метод, що відповідає за генерацію імені файлу;

Клас Storage – клас, що представляє сховище даних в оперативній пам’яті та відповідає за балансування та взаємодію з ним.

Методи класу:

– public string PerformRequest(Command command) – метод, що відповідає за вибір алгоритму дій зі сховищем в залежності від отриманої команди;

– private string PerformHSet(HSetCommand command) – метод, що реалізує послідовність дій, якщо отримано команду типу hset;

– private string PerformHGetAll(HGetAllCommand command) – метод, що реалізує послідовність дій, якщо отримано команду типу hgetall;

– private string PerformHGet(HGetCommand command) – метод, що реалізує послідовність дій, якщо отримано команду типу hget;

– private string PerformHVal(HValCommand command) – метод, що реалізує послідовність дій, якщо отримано команду типу hval;

– private string PerformHKeys(HKeysCommand command) – метод, що реалізує послідовність дій, якщо отримано команду типу hkeys;

– private string PerformHRemove(HRemoveCommand command) – метод, що реалізує послідовність дій, якщо отримано команду типу hremove;

– private string PerformLAdd(LAddCommand command) – метод, що реалізує послідовність дій, якщо отримано команду типу ladd;

– private string PerformLGetAll(LGetAllCommand command) – метод, що реалізує послідовність дій, якщо отримано команду типу lgetall;

– private string PerformLRemove(LRemoveCommand command) – метод, що реалізує послідовність дій, якщо отримано команду типу lremove;

– private string PerformLCount(LCountCommand command) – метод, що реалізує послідовність дій, якщо отримано команду типу lcount;

– private string PerformSet(SetCommand command) – метод, що реалізує послідовність дій, якщо отримано команду типу set;

– private string PerformGetAll(GetAllCommand command) – метод, що реалізує послідовність дій, якщо отримано команду типу getall;

– private string PerformGet(GetCommand command) – метод, що реалізує послідовність дій, якщо отримано команду типу get;

– private string PerformKeys(GetKeysCommand command) – метод, що реалізує послідовність дій, якщо отримано команду типу keys;

– private string PerformUpdate(UpdateCommand command) – метод, що реалізує послідовність дій, якщо отримано команду типу update;

– private string PerformRemove(RemoveCommand command) – метод, що реалізує послідовність дій, якщо отримано команду типу remove;

– private string PerformFind(FindCommand command) – метод, що реалізує послідовність дій, якщо отримано команду типу find;

– private string PerformClear(ClearCommand command) – метод, що реалізує послідовність дій, якщо отримано команду типу clear;

– private async Task<string> PerformReplay(ReplayCommand replayCommand) – метод, що відповідає за виконання нехобхідних команд під час балансування системи;

– private async Task<string> PerformSync(SyncCommand command) – метод, що відповідає за синхронізацію файлу останнього стану системи з рівнем планувальників;

3.2.3 Обробка запитів

Перед виконанням, отримані від клієнта запити перевіряються на валідність. Якщо запит є некоректним, клієнту надсилається відповідне повідомлення. У випадку коректного запиту, система визначає тип команди, яку потрібно виконати. Після цього слідує виконання команди і надсилання результату клієнту.

3.2.4 Балансування системи

Механіз балансування системи базується на використанні хеш-значень даних, які клієнт додає, модифікує чи видаляє зі сховища. В залежності від кількості доступних серверів рівня сховища, кожен з цих серверів отримує діапазон хеш-значень, з якими від буде працювати.

Послідовність дій системи при отриманні запиту від клієнта:

– Перевірка запиту на коректність;

– Визначення хеш-значення від даних, що містяться у запиті;

– Надсилання запиту на сервер, який працює з діапазоном, у який потрапляє хеш-значення даних;

– Виконання запиту;

– Надсилання відповіді клієнту;

У випадку виходу з ладу одного з серверів сховища, система виконує перебаласування. Визначаються нові актуальні діапазони для кожного сервера сховища і відбувається завантаження необхідних даних із файлу останнього стану системи.

3.2.5 Відмовостійкість системи

Відмовостійкість системи огранізована за допомогою резервного копіювання даних у файл, що знаходиться на жорсткому диску. При цьому, кожен сервер рівня планувальників та рівня сховища має власний файл останнього стану системи. Усі ці файли синхронізуються між собою для уникнення помилок у роботі системи, пов’язаних з неактуальністю даних.

У разі виникнення необхідності у перебалансування системи, файли останнього стану даних використовуються системою для відновлення даних у сховищі.

Крім того, такий підхід резервного копіювання надає можливість первинної ініціалізації сховища. Для цього необхідно перед запуском системи додати у файл початкові дані. Під час запуску система виконає відповідні команди та завантажить дані до сховища.

ВИСНОВОК ДО РОЗДІЛУ 3

У даному розділі представлено опис розробленої системи.

Під час розробки клієнтського додатку було реалізовано увесь необхідний функціонал для забезпечення коректної взаємодії з системою. Розроблений функціонал націлений лише на зміну даних у сховищі і ніяк не відноситься до основної логіки. Такий підхід забезпечує можливість переносу системи на інші операційні платформи та клієнтські додатки без особливих затрат часу.

Серверна частина була побудована з урахуванням усіх вимог, що ставляться до систем подібного типу. Реалізовано механізми обробки клієнтських запитів, реплікації останнього стану системи, балансування навантаження, роботи з різними структурами даних. Також, передбачено можливість роботи у багатокористувацькому режимі, що дозволяє обслуговувати одночасно багатьох клієнтів.

РОЗДІЛ 4
РЕЗУЛЬТАТИ ДОСЛІДНОЇ ЕКСПЛУАТАЦІЇ

У даному розділі наведено результати експлуатації системи, опис роботи клієнтської та серверної частини. Розроблена система призначена для зберігання даних в оперативній пам’яті.

4.1 Системні вимоги

– ОС: MS Windows 7/8/10, 32/64-bit;

– об’єм оперативної пам’яті: 2 Гб;

– об’єм вільного простору на жорсткому диску: 100 Мб;

– тактова частота процесора: 1,4 ГГц і вище;

4.2 Опис роботи з клієнтським додатком

Клієнтський додаток представлений у вигляді консольної програми. Після запуску, програма автоматично під’єднується до сервера рівня планувальників та інформує користувача про готовність роботи (Рис. 4.1.).

Рисунок 4.1. Інформування клієнта про готовність до роботи

Далі користувачу необхідно ввести запит до сховища. Перелік доступних запитів представлено у таблиці 4.1.

Таблиця 4.1. – Перелік доступних запитів

Запит	Опис запиту
set <ключ> <значення>	Додати до сховища пару <ключ> <значення>.
get <ключ>	Отримати зі сховища <значення>, пов’язане з ключем <ключ>.
getall	Отримати зі сховища усі ключі <ключ> та значення <значення>.
keys	Отримати зі сховища усі ключі <ключ>.
update <ключ>	Оновити значення <значення>, пов’язане з ключем <ключ>.
remove <ключ>	Видалити значення <значення>, пов’язане з ключем <ключ>.
find <ключ>	Перевірити, чи ключ <ключ> уже зайнятий.
clear	Очистити сховище.
hset <ключ> <ключ_> <значення>	Створити у сховищі хеш-таблицю, пов’язану з ключем <ключ> і додати до неї пару <ключ_> <значення>.
hget <ключ> <ключ_>	Отримати зі сховища пару <ключ_> <значення>, яка знаходиться хеш-таблиці, пов’язаній з ключем <ключ>.

Продовження таблиці 4.1. – Перелік доступних запитів	
hgetall <ключ>	Отримати зі сховища усі пари <ключ_> <значення> з хеш-таблиці, пов’язаної з ключем <ключ>.
hval <ключ>	Отримати зі сховища усі значення з хеш-таблиці, пов’язаної з ключем <ключ>.
hremove <ключ> <ключ_>	Видалити значення, пов’язане з ключем <ключ_> у хеш-таблиці з ключем <ключ>.
hkeys <ключ>	Отримати зі сховища усі ключі з хеш-таблиці, пов’язаної з ключем <ключ>.
ladd <ключ> <значення>	Створити у сховищі масив, пов’язаний з ключем <ключ> і додати до нього значення <значення>.
lgetall <ключ>	Отримати зі сховища усі значення <значення> з масиву, пов’язаного з ключем <ключ>.
lremove <ключ> <значення>	Видалити значення <значення> з масива, пов’язаного з ключем <ключ>.
lcount <ключ>	Отримати зі сховища кількість елементів масиву, пов'язаного з ключем <ключ>.

Після введення користувачем запиту, програма надсилає його до серверної частини. Зі сторони користувача взаємодія з системою здійсьюється виключно у вигляді надсилання запитів та отримання відповідей від серверної

частини. На рисунку 4.2 зображено приклад введення користувацького запиту та отримання відповіді.

Рисунок 4.2. Приклад введення користувацького запиту та отримання відповіді

У випадку коли користувач вводить некоректний запит або намагається надіслати пустий запит, програма виводить відповідне повідомлення. На рисунку 4.3 зображено приклад некорректного вводу.

Рисунок 4.3. Приклад вводу некоректного запиту

Інша проблемна ситуація виникає коли користувач намагається виконати запит, призначений для роботи з типом «рядок», над даними складних типів,

таких як «хеш-таблиця» та «масив», або у зворотньому порядку. У такому випадку програма виводить відповідне повідомлення. На рисунку 4.4 зображено приклад спроби виконати подібний запит.

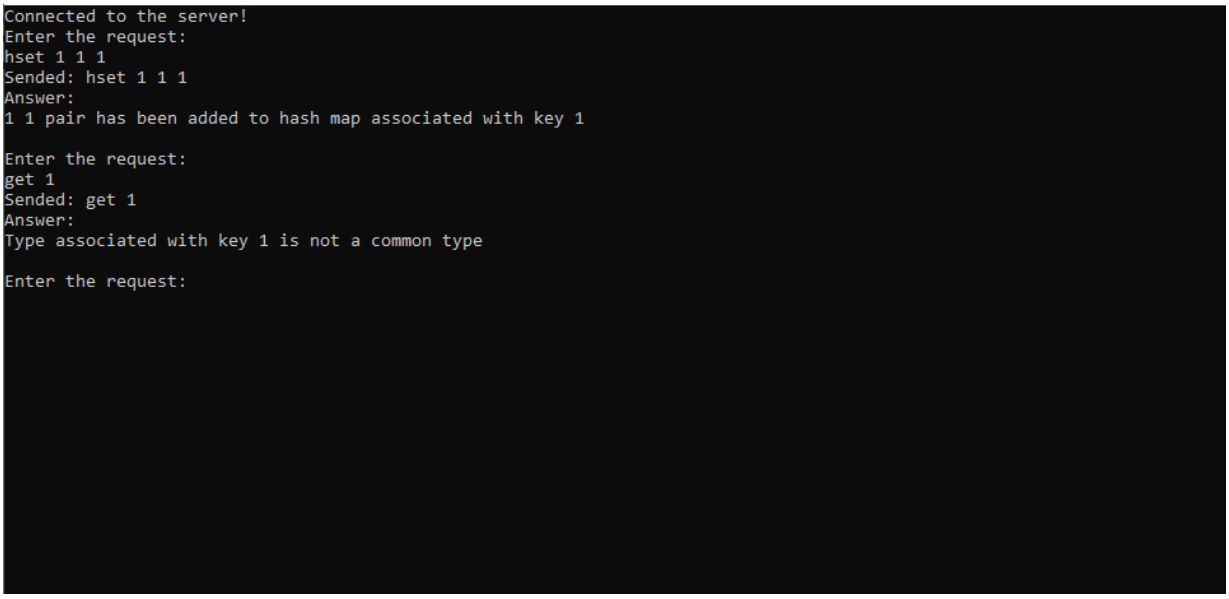


Рисунок 4.4. Приклад спроби виконати запит, призначений для роботи з даними типу «рядок», над даними типу «хеш-таблиця»

4.3 Опис роботи серверної частини

Серверна частина системи складається з рівня планувальників та рівня сховища. Після запуску, планувальник очікує підключення клієнтів та серверів сховища. У разі підключення клієнта або сервера сховища, планувальник логує це підключення і продовжує очікувати подальші підключення. Логування підключеннь до планувальника зображено на рисунку 4.5.

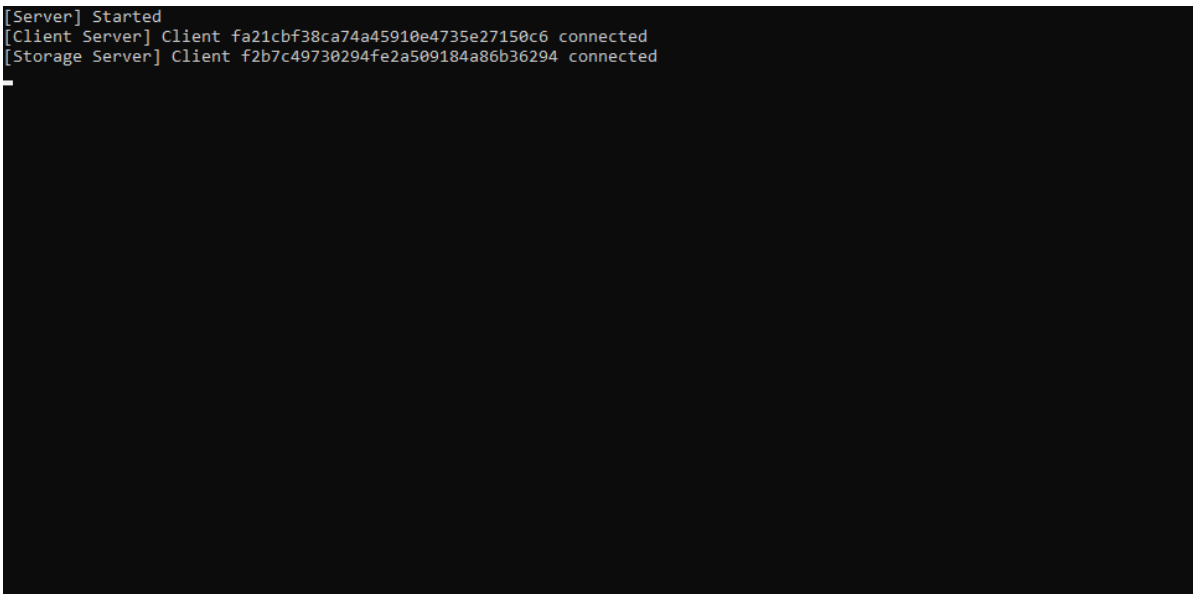


Рисунок 4.5. Логування підключень до планувальника

Під час взаємодії користувача з системою можливий варіант, коли до поточного планувальника не підключено жодного сервера сховища, тобто клієнту нікуди надсилати запити. При такій ситуаціїї планувальник надсилає клієнту відповідне повідомлення. Приклад подібної ситуації зображено на рисунку 4.6.

Рисунок 4.6. Приклад роботи системи, коли до планувальника не підключено жодного серверу сховища

Одразу після підключення серверу сховища до планувальника, відбувається синхронізація файлів останнього стану системи. Це гарантує, що кожен сервер сховища буде мати актуальну копію усього набору даних. На рисунку 4.7 забражено логування сервером сховища процесу синхронізації файлів.

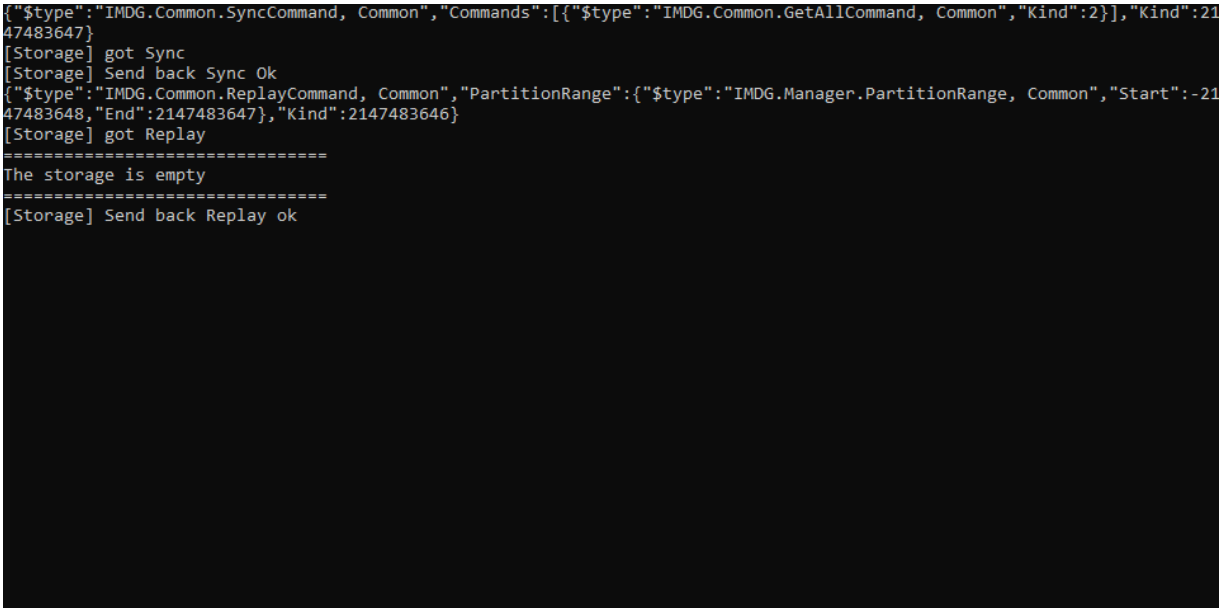


Рисунок 4.7. Лог синхронізації файлів

Синхронізація файлів останнього стану даних у сховищі відбувається кожні п’ять секунд. Таким чином забезпечується актуальність останнього стану сховища на всіх серверах кластера та планувальниках.

Для резервоного копіювання останнього стану даних у роботі було вирішено використовувати файл формату json. Такий вибір був зроблений з огляду на зручність програмної обробки файлів такого типу.

Кожного разу, коли планувальник отримує запит від клієнта, відбувається перевірка на необхідність модифікації файлу з усім набором даних. У випадках запиту на додавання, модифікацію або видалення даних файл модифікується і синхронізується файлами інших серверів.

Балансування системи відбувається кожного разу, коли під’єднується або від’єднується будь який сервер сховища. Планувальник перевизначає діапазони

хеш-значень, з якими буде працювати кожен сервер сховища та їм обчислені діапазони. Далі, сервери завантажують необхідні дані з файлу до своєї оперативної пам’яті. Такий підхід гарантує, що дані будуть доступні навіть якщо вийшов з ладу сервер, на якому вони знаходилися. На рисунку 4.8 зображено логування сервером сховища процесу балансування.

Рисунок 4.8. Лог сервера сховища під час балансування

Використовуючи файл останнього стану даних можна провести первинну ініціалізацію сховища даними. Для цього необхідно занести до файлу дані, які повинні бути завантажені при запуску системи. Під час запуску сховище буде збалансоване і дані з файлу будуть завантажені на сервери кластера. Цей формат На рисунках 4.9 – 4.11 зображено приклад первинної ініціалізації сховища та взаємодія користувача з ними.

```
{"$type":"IMDG.Common.HSetCommand, Common","Field":"Hanks","Kind":8,"Value":"Tom","Key":"1"}
{"$type":"IMDG.Common.HSetCommand, Common","Field":"Cruise","Kind":8,"Value":"Tom","Key":"1"}
{"$type":"IMDG.Common.HSetCommand, Common","Field":"Smith","Kind":8,"Value":"Will","Key":"1"}
{"$type":"IMDG.Common.HSetCommand, Common","Field":"Willis","Kind":8,"Value":"Bruce","Key":"1"}
{"$type":"IMDG.Common.HSetCommand, Common","Field":"Stathem","Kind":8,"Value":"Jason","Key":"1"}
{"$type":"IMDG.Common.HSetCommand, Common","Field":"Oldman","Kind":8,"Value":"Gary","Key":"1"}
```

Рисунок 4.9. Вміст файлу останнього стану даних

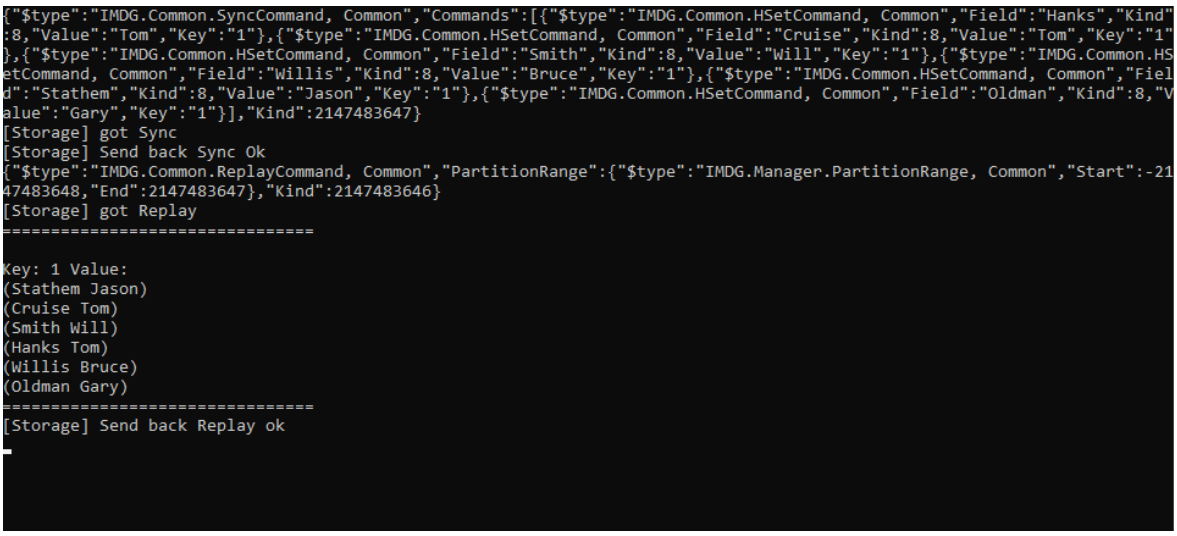


Рисунок 4.10. Завантаження даних з файлу під час балансування сховища

Рисунок 4.11. Використання користувачем даних, які були завантажені при первинній ініціалізації

ВИСНОВОК ДО РОЗДІЛУ 4

У даному розділі представлено результати дослідної експлуатації та опис роботи клієнтської та серверної частин.

Клієнтський додаток є досить простим у користуванні, тому у користувачів не повинно виникати труднощів під час його використання. За результатами експлуатації можна зробити висновок, що зі своєї сторони клієнтська програма коректно обробляє усі проблемні ситуації, які можуть призвести до помилок у роботі системи. Крім того, оскільки уся основна логіка міститься у серверній частині, система може використовуватися різними типами клієнтів.

У серверній частині реалізовані всі ключові аспекти, а саме підтримка складних типів даних, можливість роботи з великою кількістю користувачів, відмовостійкість, масштабованість. Виходячи з результатів дослідної експлуатації можна зробити висновок, що усі з перерахованих механізмів працюють коректно і задовольняють вимоги, які ставляться до систем поддібного типу.

ВИСНОВКИ

Дана робота складається з чотирьох розділів. У першому розділі проведено теоретичний аналіз реалізацій розподілених сховищ даних в пам’яті, розглянуто переваги та недоліки таких систем, наведено порівняльну характеристику існуючих систем.

У другому розділі були запропоновані найбільш оптимальні архітектурні рішення для побудови системи, описано клієнтську та серверну частини, представлено загальну структуру системи з урахуванням відмовостійкості та балансування навантаження. Розділ містить обгрунтування щодо вибору середовища розробки та мови програмування.

У третьому розділі наведено стислий опис розробленої системи, розглянуто перелік модулів клієнтської та серверної частини. Перераховано користувацькі функції, створені для забезпечення коректного функціонування системи.

У четвертому розділі наведено опис роботи клієнтської та серверної частини. Перевірено коректність роботи механізмів перевірки запитів, резервного копіювання, балансування навантаження, первинної ініціалізації. Представлено рисунки, зображають дії системи при різних проблемних ситуаціях.

Результатом роботи є система для зберігання даних в оперативній пам’яті. Система розроблена з урахуванням питань масштабованості, відмовостійкості, паралельної обробки великої кількості запитів. У системі повністю відсутні пересилання даних між серверами сховища, що у свою чергу дозволяє уникнути зниження швидкості роботи.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Подрубайло А.А. Повышение производительности хранилищ данных в оперативной памяти посредством использования программной транзакционной памяти. URL: https://cyberleninka.ru/article/n/povyshenie-proizvoditelnosti-hranilisch-dannyh-v-operativnoy-pamyati-posredstvom-ispolzovaniya-programmnoy-tranzaktsionnoy-pamyati

2. Бузовский О.В., Подрубайло А.А. Методы и алгоритмы обьеденинения таблиц для распределенных хранилищ данных в оперативной памяти. URL: http://it-visnyk.kpi.ua/wp-content/uploads/2015/03/issue-60_p74-83.pdf

3. Бузовский О.В., Подрубайло А.А. Актуальные проблемы распределенных хранилищ данных в оперативной памяти. Вісник Національного авіаційного університету, 2015, 36-43 сс.

4. System Properties Comparison GridGain vs. Hazelcast vs. NCache vs.Redis. URL: https://db-engines.com/en/system/GridGain%3BHazelcast%3BNCache%3BRedis

5. Ivanov N. GridGain, In-Memory Data Grid: Explained..., 2012. URL: https://www.gridgain.com/resources/blog/in-memory-data-grid-explained

6. Roy Prins. In-Memory Data Grids, 2013. URL: https://dzone.com/articles/memory-data-grids

7. Что такое In-Memory Data Grid. URL: https://habrahabr.ru/post/160517

8. О модели взаимодействия клиент-сервер. Архитектура «клиент-сервер» с примерами. URL: https://zametkinapolyah.ru/servera-i-protokoly/o-modeli-vzaimodejstviya-klient-server-prostymi-slovami-arxitektura-klient-server-s-primerami.html

9. Клієнт-серверні системи. URL: https://bit.ly/2EbTwsF

10. What is Client/Server Architecture? - Definition from Techopedia. URL: https://www.techopedia.com/definition/438/clientserver-architecture

11. Документация по .NET Core. URL: https://docs.microsoft.com/ru-ru/dotnet/core/

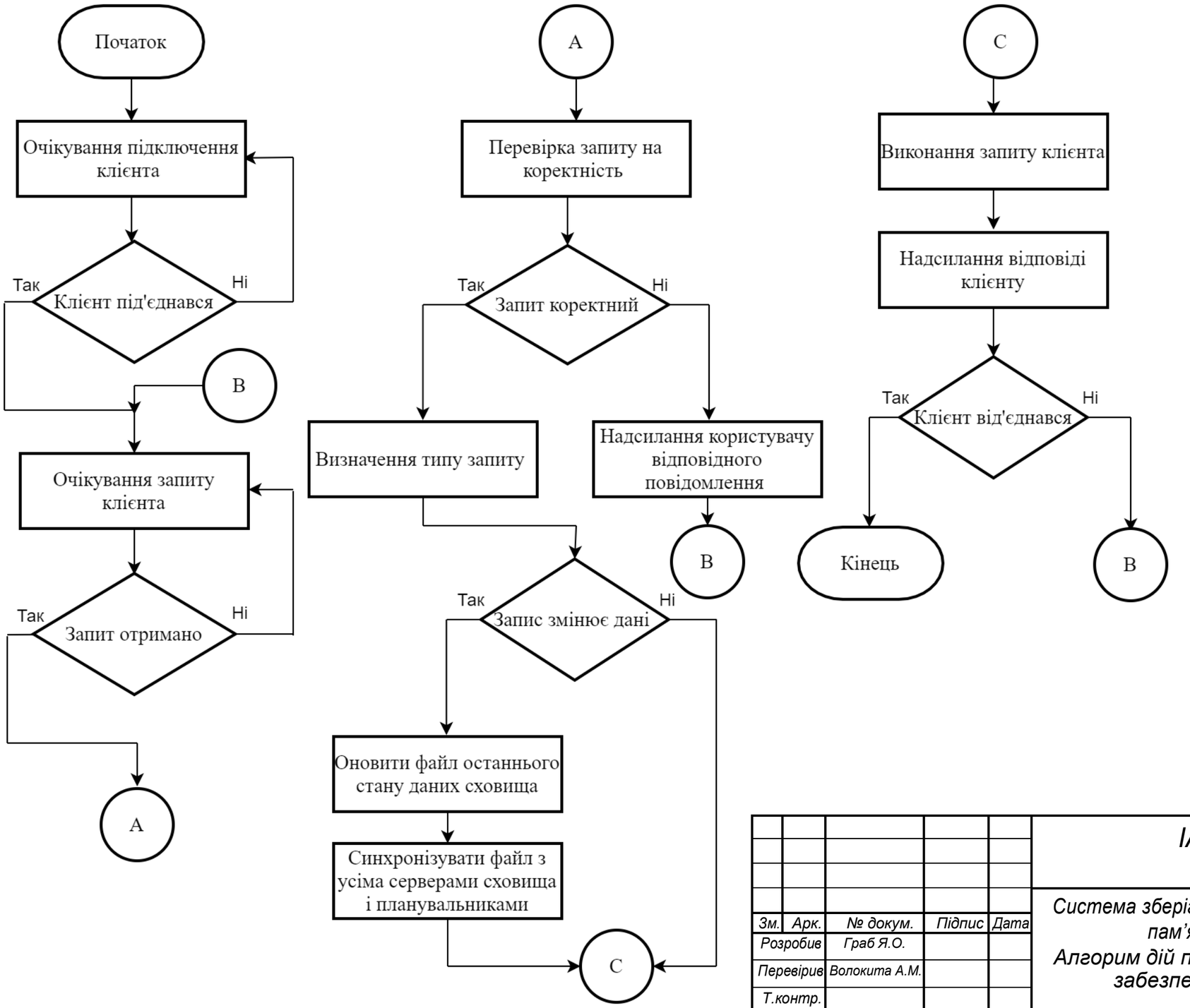
ДОДАТОК 1

Система зберігання даних в пам’яті

Алгоритм дій програмного забезпечення

ІАЛЦ.467100.004 Д1

Київ – 2020



					ІАЛЦ.467100.004 Д1	
Зм.	Арк.	№ докум.	Підпис	Дата	Система зберігання даних в пам'яті Алгорим дій програмного забезпечення	Літера / Маса / Масштаб
Розробив		Граб Я.О.				
Перевірив		Волокита А.М.				Аркуш 1 / Аркушів 1
Т.контр.						
					Дипломний проект	НТУУ "КПІ Ім Ігоря Сікорського", ФІОТ, гр. ІП-64
Н.контр.		Сімоненко В.П.				
Затверд.						

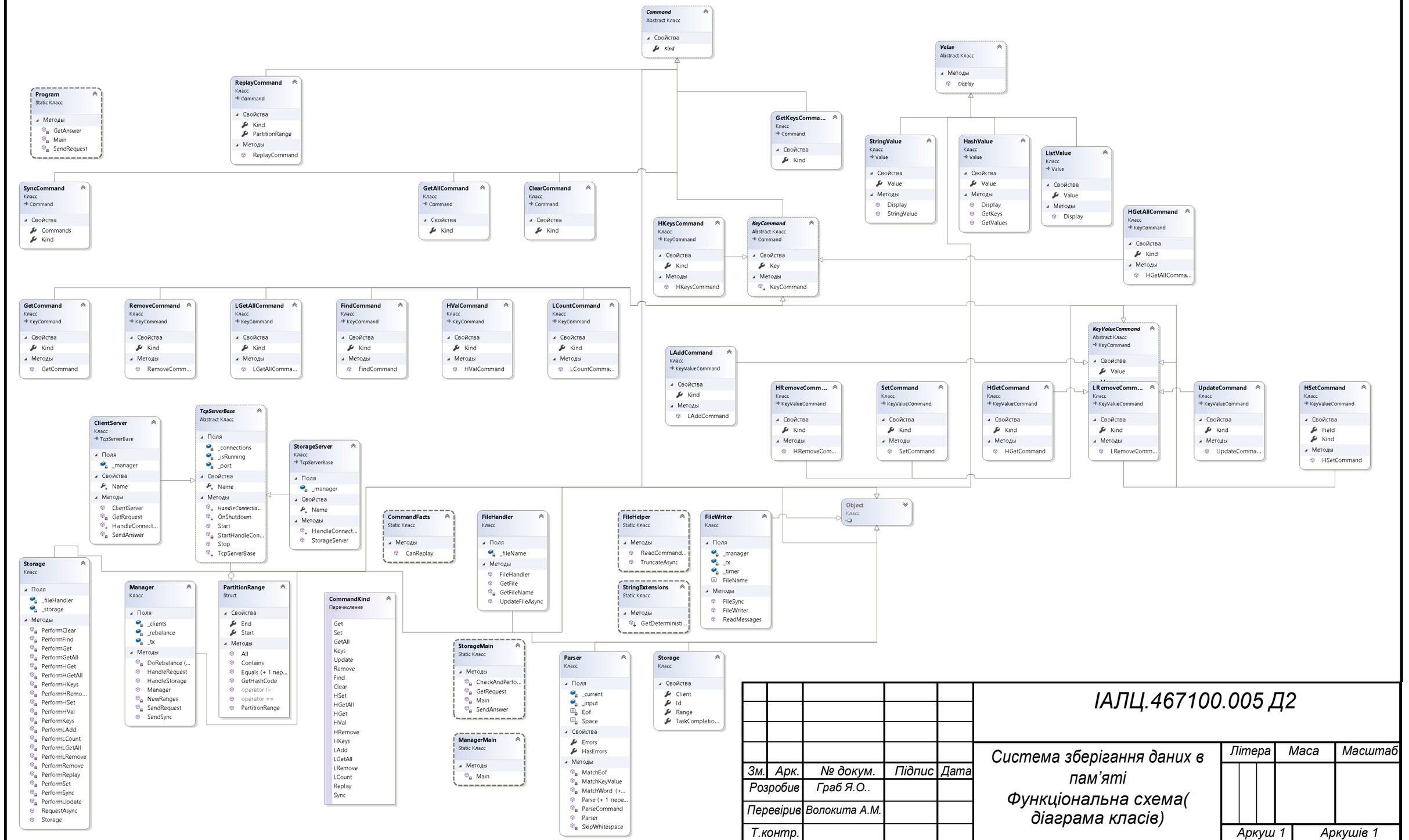
ДОДАТОК 2

Система зберігання даних в пам’яті

Структурна схема (діаграма класів)

ІАЛЦ.467100.005 Д2

Київ – 2020



					ІАЛЦ.467100.005 Д2
Зм.	Арк.	№ докум.	Підпис	Дата	
Розробив		Граб Я.О..			Система зберігання даних в пам’яті Функціональна схема(діаграма класів)
Перевірив		Волокита А.М.			
Т.контр.					
Н.контр.		Сімоненко В.П.			Дипломний проект
Затверд.					

Літера	Маса	Масштаб
Аркуш 1	Аркушів 1	

НТУУ "КПІ Ім Ігоря Сікорського", ФІОТ, гр. ІП-64

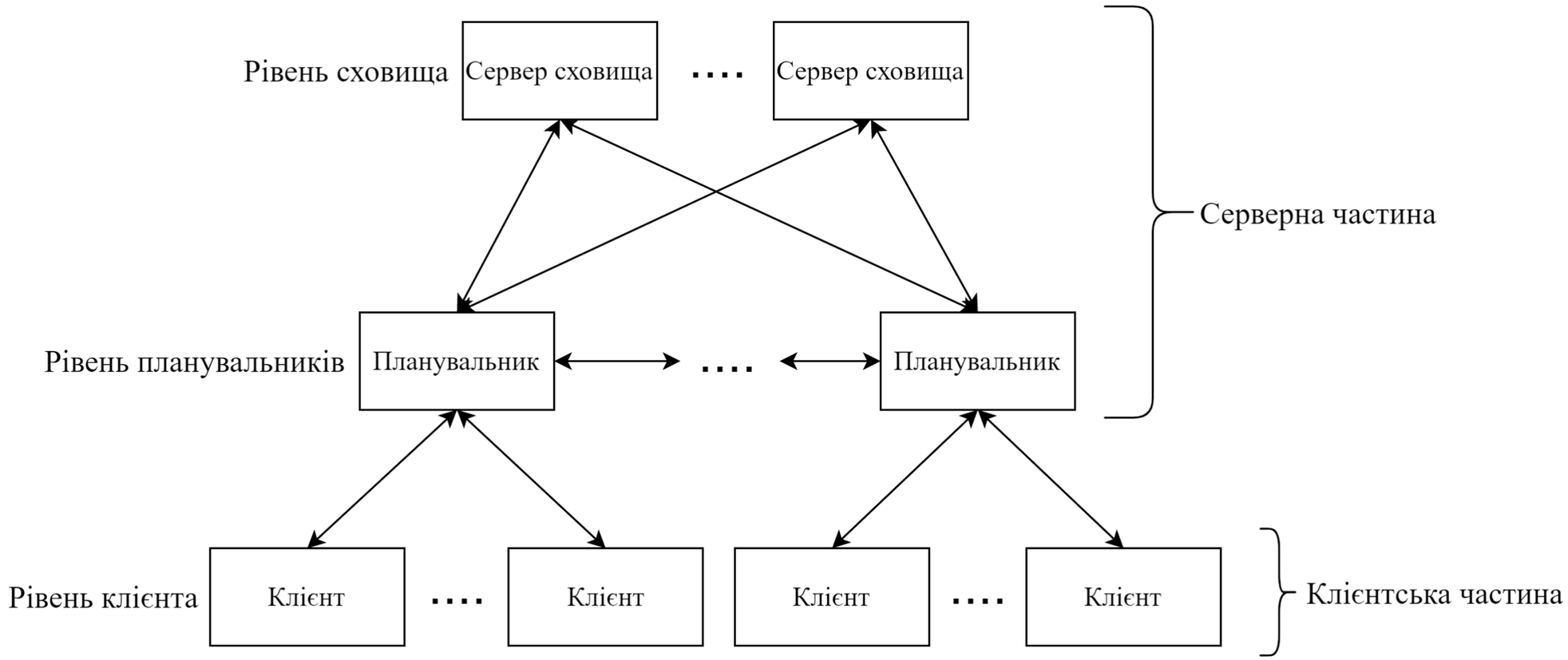
ДОДАТОК З

Система зберігання даних в пам’яті

Структурна схема

ІАЛЦ.467100.006 ДЗ

Київ – 2020



					ІАЛЦ.467100.006 ДЗ	
Зм.	Арк.	№ докум.	Підпис	Дата	Система зберігання даних в пам'яті Структурна схема	Літера / Маса / Масштаб
Розробив		Граб Я.О.				
Перевірив		Волокита А.М.				Аркуш 1 / Аркушів 1
Т.контр.						
Н.контр.		Сімоненко В.П.			Дипломний проєкт	НТУУ "КПІ Ім Ігоря Сікорського", ФІОТ, гр. ІП-64
Затверд.						

ДОДАТОК 4

Система зберігання даних в пам’яті

Тексти програмного коду

ІАЛЦ.467100.007 Д4

Київ – 2020

ClientMain.cs

```
using System;
using System.Buffers;
using System.IO;
using System.Net.Sockets;
using System.Text;
using System.Threading.Tasks;

namespace IMDG.Client
{
    public static class ClientMain
    {
       static async Task Main(string[] args)
        {
            const int port = 7000;
            var pool = MemoryPool<byte>.Shared;
            using var byteRead = pool.Rent(1024);
            using var byteWrite = pool.Rent(1024);
            try
            {
                using var client = new TcpClient("127.0.0.1", port);
                await using var stream = client.GetStream();
                Console.WriteLine("Connected to the server!");
                while (true)
                {
                    Console.WriteLine("Enter the request:");
                    var request = Console.ReadLine();
                    if (request == "")
                    {
                        Console.WriteLine("No request to send!\n");
                        continue;
                    }
                    await SendRequest(stream, request, byteWrite.Memory);
                    var answer = await GetAnswer(stream, byteRead.Memory);
                }

                //client.Close();
                //Console.WriteLine("Client closed the connection");
            }
            catch (Exception e)
            {
                Console.WriteLine(e.Message);
            }

        }
        private static async Task<string> GetAnswer(Stream stream, Memory<byte> byteRead)
        {
            var length = await stream.ReadAsync(byteRead);
            var answer = Encoding.UTF8.GetString(byteRead.Span.Slice(0, length));
            Console.WriteLine("Answer: " + "\n" + answer);
            return answer;
        }
        private static async Task SendRequest(Stream stream, string request, Memory<byte>
byteWrite)
        {
            int len = Encoding.UTF8.GetBytes(request, byteWrite.Span);
            await stream.WriteAsync(byteWrite.Slice(0, len));
            await stream.FlushAsync();
            Console.WriteLine("Sended: " + request);
        }

    }
}
```

Command.cs

```
using System.Collections.Generic;
using IMDG.Manager;

namespace IMDG.Common
{
    public abstract class Command
    {
        public abstract CommandKind Kind { get; }
    }

    public class SyncCommand : Command
    {
        public IList<Command> Commands { get; set; }
        public override CommandKind Kind => CommandKind.Sync;
    }

    public class ReplayCommand : Command
    {
        public PartitionRange PartitionRange { get; }

        public ReplayCommand(PartitionRange partitionRange) => PartitionRange =
partitionRange;

        public override CommandKind Kind => CommandKind.Replay;
    }

    public abstract class KeyCommand : Command
    {
        protected KeyCommand(string key)
        {
            Key = key;
        }

        public string Key { get; }
    }

    public abstract class KeyValueCommand : KeyCommand
    {
        protected KeyValueCommand(string key, string value) : base(key)
        {
            Value = value;
        }

        public string Value { get; }
    }

    public class GetCommand : KeyCommand
    {
        public GetCommand(string key) : base(key)
        {
        }

        public override CommandKind Kind => CommandKind.Get;
    }

    public class SetCommand : KeyValueCommand
    {
        public SetCommand(string key, string value) : base(key, value)
        {
        }

        public override CommandKind Kind => CommandKind.Set;
    }
```

```
public class GetAllCommand : Command
{
    public override CommandKind Kind => CommandKind.GetAll;
}

public class UpdateCommand : KeyValueCommand
{
    public override CommandKind Kind => CommandKind.Update;

    public UpdateCommand(string key, string value) : base(key, value)
    {
    }
}

public class GetKeysCommand : Command
{
    public override CommandKind Kind => CommandKind.Keys;
}

public class RemoveCommand : KeyCommand
{
    public RemoveCommand(string key) : base(key)
    {
    }

    public override CommandKind Kind => CommandKind.Remove;
}

public class FindCommand : KeyCommand
{
    public FindCommand(string key) : base(key)
    {
    }

    public override CommandKind Kind => CommandKind.Find;
}

public class ClearCommand : Command
{
    public override CommandKind Kind => CommandKind.Clear;
}

public class HSetCommand : KeyValueCommand
{
    public string Field { get; }
    public HSetCommand(string key, string field, string value) : base(key, value)
    {
        Field = field;
    }

    public override CommandKind Kind => CommandKind.HSet;
}
public class HGetAllCommand : KeyCommand
{
    public HGetAllCommand(string key) : base(key)
    {
    }

    public override CommandKind Kind => CommandKind.HGetAll;
}
public class HRemoveCommand : KeyValueCommand
{
    public HRemoveCommand(string key, string value) : base(key, value)
    {
```

```
        }
        public override CommandKind Kind => CommandKind.HRemove;
    }
    public class HGetCommand : KeyValueCommand
    {
        public HGetCommand(string key, string value) : base(key, value)
        {
        }

        public override CommandKind Kind => CommandKind.HGet;
    }
    public class HValCommand : KeyCommand
    {
        public HValCommand(string key) : base(key)
        {
        }

        public override CommandKind Kind => CommandKind.HVal;
    }

    public class HKeysCommand : KeyCommand
    {
        public HKeysCommand(string key) : base(key)
        {
        }

        public override CommandKind Kind => CommandKind.HKeys;
    }
    public class LAddCommand : KeyValueCommand
    {
        public LAddCommand(string key, string value) : base(key, value)
        {
        }

        public override CommandKind Kind => CommandKind.LAdd;
    }
    public class LGetAllCommand : KeyCommand
    {
        public LGetAllCommand(string key) : base(key)
        {
        }

        public override CommandKind Kind => CommandKind.LGetAll;
    }
    public class LRemoveCommand : KeyValueCommand
    {
        public LRemoveCommand(string key, string value) : base(key, value)
        {
        }
        public override CommandKind Kind => CommandKind.LRemove;
    }
    public class LCountCommand : KeyCommand
    {
        public LCountCommand(string key) : base(key)
        {
        }

        public override CommandKind Kind => CommandKind.LCount;
    }
}
```

CommandFacts.cs

```
using System;

namespace IMDG.Common
{
    public static class CommandFacts
    {


        public static bool CanReplay(Command command) =>
            command.Kind switch
            {
                CommandKind.Get => false,
                CommandKind.Set => true,
                CommandKind.GetAll =>  false,
                CommandKind.Keys => false,
                CommandKind.Update =>  true,
                CommandKind.Remove => true,
                CommandKind.Find => false,
                CommandKind.Clear => true,
                CommandKind.HSet => true,
                CommandKind.HGetAll => false,
                CommandKind.HGet => false,
                CommandKind.HVal => false,
                CommandKind.HRemove => true,
                CommandKind.HKeys => false,
                CommandKind.LAdd => true,
                CommandKind.LGetAll => false,
                CommandKind.LRemove => true,
                CommandKind.LCount => false,
                CommandKind.Replay => false,
                CommandKind.Sync => false,
                _ => throw new ArgumentOutOfRangeException(nameof(command.Kind))
            };
    }
}
```

CommandKind.cs

```
using System;

namespace IMDG.Common
{
    public enum CommandKind
    {
        Get,
        Set,
        GetAll,
        Keys,
        Update,
        Remove,
        Find,
        Clear,
        HSet,
        HGetAll,
        HGet,
        HVal,
        HRemove,
        HKeys,
        LAdd,
        LGetAll,
        LRemove,
        LCount,
        Replay = int.MaxValue - 1,
```

```
        Sync = int.MaxValue,
    }
}
```

FileHelper.cs

```
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Newtonsoft.Json;

namespace IMDG.Common
{
    public static class FileHelper
    {
        public static async Task<IEnumerable<Command>> ReadCommandsFromFile(string fileName,
Encoding encoding = null)
        {
            if (!File.Exists(fileName)) return Enumerable.Empty<Command>();
            var lines = await File.ReadAllLinesAsync(fileName, encoding ?? Encoding.UTF8);
            return lines
                .Where(l => !string.IsNullOrWhiteSpace(l))
                .Select(l => l.TrimEnd())
                .Select(l =>
                    JsonConvert.DeserializeObject<Command>(l, new JsonSerializerSettings
                    {
                        TypeNameHandling = TypeNameHandling.Objects
                    }));
        }

        public static async Task TruncateAsync(string fileName, IEnumerable<Command>
commands, Encoding encoding = null)
        {
            var lines = commands.Select(c => JsonConvert.SerializeObject(c, new
JsonSerializerSettings
            {
                TypeNameHandling = TypeNameHandling.Objects
            }));
            await File.WriteAllLinesAsync(fileName, lines, encoding ?? Encoding.UTF8);
        }
    }
}
```

PartitionRange.cs

```
using System;

namespace IMDG.Manager
{
    /// <summary>
    ///
    /// </summary>
    /// <remarks>Inclusive at both ends</remarks>
    public readonly struct PartitionRange : IEquatable<PartitionRange>
    {
        public int Start { get; }
        public int End { get; }


        public PartitionRange(int start, int end)
        {
            Start = start;
            End = end;
```

```
        }

        public static PartitionRange All()
        {
            return new PartitionRange(int.MinValue, int.MaxValue);
        }


        public bool Equals(PartitionRange other) => Start == other.Start && End ==
other.End;

        public override bool Equals(object obj) => obj is PartitionRange other &&
Equals(other);

        public override int GetHashCode() => HashCode.Combine(Start, End);

        public static bool operator ==(PartitionRange left, PartitionRange right) =>
left.Equals(right);

        public static bool operator !=(PartitionRange left, PartitionRange right) =>
!left.Equals(right);

        public bool Contains(int value) => value >= Start && value <= End;
    }
}
```

StringExtensions.cs

```
namespace IMDG.Common
{
    public static class StringExtensions
    {
       public static int GetDeterministicHashCode(this string str)
        {
            unchecked
            {
                int hash1 = (5381 << 16) + 5381;
                int hash2 = hash1;

                for (int i = 0; i < str.Length; i += 2)
                {
                    hash1 = ((hash1 << 5) + hash1) ^ str[i];
                    if (i == str.Length - 1)
                        break;
                    hash2 = ((hash2 << 5) + hash2) ^ str[i + 1];
                }

                return hash1 + (hash2 * 1566083941);
            }
        }

    }
}
```

Value.cs

```
using System.Collections.Concurrent;
using System.Collections.Generic;
using System.Linq;
using System.Text.Json.Serialization;
using Newtonsoft.Json;

namespace IMDG.Common
{
    public abstract class Value
```

```
    {
        public abstract string Display();
    }


    public class StringValue : Value
    {
        public string Value { get; set; }

        public StringValue(string value)
        {
            Value = value;
        }

        public override string Display() => Value;
    }

    public class HashValue : Value
    {
        public ConcurrentDictionary<string, string> Value { get; set; } = new
ConcurrentDictionary<string, string>();

        public override string Display() =>
            Value.Aggregate(string.Empty,
                (current, keyValue) => current + $"\n({keyValue.Key}" + " " +
$"{keyValue.Value})");
        public string GetValues() =>
            Value.Aggregate(string.Empty,
                (current, keyValue) => current + $"\n({keyValue.Value})");
        public string GetKeys() =>
            Value.Aggregate(string.Empty,
                (current, keyValue) => current + $"\n({keyValue.Key})");
    }

    public class ListValue : Value
    {
        public List<string> Value { get; set; } = new List<string>();

        public override string Display() =>
            Value.Aggregate(string.Empty,(current,value)=> current + $"{value}" + " ");
    }
}
```

ClientServer.cs

```
using System;
using System.Buffers;
using System.IO;
using System.Net.Sockets;
using System.Text;
using System.Threading.Tasks;

namespace IMDG.Manager
{
    public class ClientServer : TcpServerBase
    {
        private readonly Manager _manager;


        public ClientServer(Manager manager, int port = 7000) : base(port)
        {
            _manager = manager;
        }

        protected override string Name => "Client Server";
```

```
        protected override async Task HandleConnectionAsync(TcpClient tcpClient, Guid _)
        {
            await using var stream = tcpClient.GetStream();
            var pool = MemoryPool<byte>.Shared;
            using var byteRead = pool.Rent(100 * 1024);
            using var byteWrite = pool.Rent(100 * 1024);
            while (tcpClient.Connected)
            {
                var request = await GetRequest(stream, byteRead.Memory);
                var handleRequest = await _manager.HandleRequest(request) + '\n';
                await SendAnswer(stream, handleRequest, byteWrite.Memory);
            }
        }

        private static async Task<string> GetRequest(Stream stream, Memory<byte> byteRead)
        {
            var length = await stream.ReadAsync(byteRead);
            var request = Encoding.UTF8.GetString(byteRead.Span.Slice(0, length));
            return request;
        }

        private static async Task SendAnswer(Stream stream, string answer, Memory<byte>
byteWrite)
        {
            int len = Encoding.UTF8.GetBytes(answer, byteWrite.Span);
            await stream.WriteAsync(byteWrite.Slice(0, len));
            await stream.FlushAsync();
        }
    }
}
```

FileWriter.cs

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Channels;
using System.Threading.Tasks;
using IMDG.Common;
using IMDG.Parser;
using Newtonsoft.Json;

namespace IMDG.Manager
{
    public class FileWriter
    {
        public const string FileName = "file.json";

        private readonly ChannelReader<Command> _rx;
        private readonly Manager _manager;

        private Timer _timer;

        public FileWriter(ChannelReader<Command> rx, Manager manager)
        {
            _rx = rx;
            _manager = manager;
            _timer = new Timer(_ => FileSync(), null, TimeSpan.FromSeconds(10),
TimeSpan.FromSeconds(5));
        }
```

```
        public Task ReadMessages()
        {
            return Task.Run(async () =>
            {
                await foreach (var command in _rx.ReadAllAsync())
                {
                    var line = JsonConvert.SerializeObject(command, new
JsonSerializerSettings
                    {
                        TypeNameHandling = TypeNameHandling.Objects
                    });
                    await File.AppendAllLinesAsync(FileName,new[] {line }, Encoding.UTF8);
                }
            });
        }

        public async void FileSync()
        {
            var commands = await FileHelper.ReadCommandsFromFile(FileName);
            await _manager.SendSync(commands);
        }

    }
}
```

Manager.cs

```
using System;
using System.Buffers;
using System.Collections.Concurrent;
using System.Collections.Generic;
using System.Collections.Immutable;
using System.Linq;
using System.Net.Sockets;
using System.Text;
using System.Threading;
using System.Threading.Channels;
using System.Threading.Tasks;
using IMDG.Common;
using IMDG.Parser;
using Newtonsoft.Json;

namespace IMDG.Manager
{
    public class Storage
    {
        public Guid Id { get; set; }
        public TcpClient Client { get; set; }

        public TaskCompletionSource<bool> TaskCompletionSource { get; set; }
        public PartitionRange Range { get; set; }
    }

    public class Manager
    {
        private readonly ChannelWriter<Command> _tx;

        private readonly ConcurrentDictionary<Guid, Storage> _clients =
            new ConcurrentDictionary<Guid, Storage>();

        public Manager(ChannelWriter<Command> tx)
        {
            _tx = tx;
        }
```

```
        private static IList<PartitionRange> NewRanges(int count)
        {
            if (count < 0) throw new ArgumentException("", nameof(count));
            if(count == 0) return new List<PartitionRange>();
            const long max = (long) int.MaxValue;
            const long min = (long) int.MinValue;
            var n = (Math.Abs(min) + max);
            var ranges = new List<PartitionRange>(count);
            for (int i = 0; i < count - 1; i++)
            {
                var lower = i * n / count;
                var upper = (i + 1) * n / count;

                ranges.Add(new PartitionRange((int)(lower - min), (int)(upper - min - 1)));
            }
            ranges.Add(new PartitionRange((int) ((count - 1) * n / count - min),
int.MaxValue));
            return ranges;
        }

        private SemaphoreSlim _rebalance = new SemaphoreSlim(1);

        public async Task HandleStorage(TcpClient client, Guid id)
        {
            await DoRebalance(client, id);
           await _clients[id].TaskCompletionSource.Task;
        }

        private async Task DoRebalance(TcpClient client, Guid id)
        {
            await _rebalance.WaitAsync();
            var newRanges = NewRanges(_clients.Count + 1);
            foreach (var (storage, newRange) in _clients.Values.Zip(newRanges))
            {
                storage.Range = newRange;
            }

            var tcs = new TaskCompletionSource<bool>();
            _clients[id] = new Storage
            {
                Id = id,
                Client = client,
                TaskCompletionSource = tcs,
                Range = newRanges[^1]
            };
            var commands = (await
FileHelper.ReadCommandsFromFile(FileWriter.FileName)).ToList();

            _rebalance.Release();
            foreach (var c in _clients.Values)
            {
                await SendRequest(new SyncCommand
                {
                    Commands = commands
                }, c);
                await SendRequest(new ReplayCommand(c.Range), c);
            }


        }

        private async Task DoRebalance()
        {
            await _rebalance.WaitAsync();
```

```
            var newRanges = NewRanges(_clients.Count);
            foreach (var (storage, newRange) in _clients.Values.Zip(newRanges))
            {
                storage.Range = newRange;
            }
            var commands = (await
FileHelper.ReadCommandsFromFile(FileWriter.FileName)).ToList();
            _rebalance.Release();
            foreach (var c in _clients.Values)
            {
                await SendRequest(new SyncCommand
                {
                    Commands = commands
                }, c);
                await SendRequest(new ReplayCommand(c.Range), c);
            }
        }

        public async Task<string> HandleRequest(string req)
        {
            await _rebalance.WaitAsync();
            _rebalance.Release();
            if (_clients.IsEmpty)
            {
                await Console.Error.WriteLineAsync("No storage");
                // TODO: Handle no clients
                return "No storages connected";
            }


            var parser = new Parser.Parser(req);
            var command = parser.Parse();
            if (parser.HasErrors)
            {
                var join = string.Join('\n', parser.Errors);
                return join;
            }

            await _tx.WriteAsync(command);
            if (command is KeyCommand kc)
            {
                var keyHash = kc.Key.GetDeterministicHashCode();
                var storage = _clients.Values.First(v => v.Range.Contains(keyHash));
                return await SendRequest(command, storage);
            }

            var sb = new StringBuilder();

            foreach (var (_, client) in _clients)
            {
                sb.AppendJoin('\n', await SendRequest(command, client));
            }

            return sb.ToString();
        }

        private async Task<string> SendRequest(Command req, Storage client)
        {
            try
            {
                // NOTE: don't dispose
                var networkStream = client.Client.GetStream();
                using var writeMemoryOwner = MemoryPool<byte>.Shared.Rent(100 * 1024);
                var writeMemory = writeMemoryOwner.Memory;
                using var readMemoryOwner = MemoryPool<byte>.Shared.Rent(100 * 1024);
```

```
                var readMemory = readMemoryOwner.Memory;


                var ser = JsonConvert.SerializeObject(req, new JsonSerializerSettings
                {
                    TypeNameHandling = TypeNameHandling.Objects
                });
                var bytes = Encoding.UTF8.GetBytes(ser, writeMemory.Span);
                await networkStream.WriteAsync(writeMemory[..bytes]);
                await networkStream.FlushAsync();
                var sb = new StringBuilder();
                int length = await networkStream.ReadAsync(readMemory);
                var answer = Encoding.UTF8.GetString(readMemory.Span[..length]);
                sb.Append(answer);
                return sb.ToString();
            }
            catch (Exception e)
            {
                Console.WriteLine(e.GetType());
                Console.WriteLine($"Storage {client.Id:N} not available");
                client.TaskCompletionSource.SetResult(true);
                _clients.Remove(client.Id, out _);
                await DoRebalance();
                return "Error: Client not available";
            }
        }

        public async Task SendSync(IEnumerable<Command> commands)
        {
            var sync = new SyncCommand
            {
                Commands = commands.ToList()
            };
            foreach (var storage in _clients.Values)
            {
                await SendRequest(sync, storage);
            }
        }
    }
}
```

ManagerMain.cs

```
using System;
using System.Threading.Channels;
using System.Threading.Tasks;
using IMDG.Common;

namespace IMDG.Manager
{
    public static class ManagerMain
    {
        private static async Task Main()
        {
            Console.WriteLine("[Server] Started");
            var ch = Channel.CreateBounded<Command>(new BoundedChannelOptions(1024)
            {
                SingleReader = true,
                SingleWriter = false,
                FullMode = BoundedChannelFullMode.Wait,
            });
            var manager = new Manager(ch.Writer);
            var fw = new FileWriter(ch.Reader, manager);
            var storageServer = new StorageServer(manager);
            var clientServer = new ClientServer(manager);
```

```
            await Task.WhenAny(
                clientServer.Start(),
                storageServer.Start(),
                fw.ReadMessages());
        }
    }
}
```

StorageServer.cs

```
using System;
using System.Net.Sockets;
using System.Threading.Tasks;

namespace IMDG.Manager
{
    public class StorageServer : TcpServerBase
    {
        private readonly Manager _manager;

        public StorageServer(Manager manager, int port = 7001) : base(port)
        {
            _manager = manager;
        }


        protected override string Name => "Storage Server";

        protected override Task HandleConnectionAsync(TcpClient tcpClient, Guid clientId)
        {
            return _manager.HandleStorage(tcpClient, clientId);
        }
    }
}
```

TcpServerBase.cs

```
using System;
using System.Collections.Generic;
using System.Net.Sockets;
using System.Threading.Tasks;

namespace IMDG.Manager
{
    public abstract class TcpServerBase
    {
        private readonly List<Task> _connections = new List<Task>();
        private readonly int _port;
        private bool _isRunning;

        protected TcpServerBase(int port = 7000)
        {
            _port = port;
        }


        protected virtual string Name { get; } = "Server";

        public async Task Start()
        {
            var tcpListener = TcpListener.Create(_port);
            tcpListener.Start();
            _isRunning = true;
            while (_isRunning)
            {
```

```
                var tcpClient = await tcpListener.AcceptTcpClientAsync();
                var id = Guid.NewGuid();
                Console.WriteLine($"[{Name}] Client {id:N} connected");
                // launch new thread
                var task = StartHandleConnectionAsync(tcpClient, id);
                // if already faulted, re-throw any error on the calling context
                if (task.IsFaulted)
                    await task;
            }
        }

        public void Stop()
        {
            _isRunning = false;
        }

        protected abstract Task HandleConnectionAsync(TcpClient tcpClient, Guid clientId);

        protected virtual void OnShutdown(Guid clientId)
        {
        }

        // Register and handle the connection
        private async Task StartHandleConnectionAsync(TcpClient tcpClient, Guid clientId)
        {
            // start the new connection task
            var connectionTask = Task.Run(() => HandleConnectionAsync(tcpClient, clientId));

            // add it to the list of pending task
            lock (_connections)
            {
                _connections.Add(connectionTask);
            }
            // catch all errors of HandleConnectionAsync
            try
            {
                await connectionTask;
                // we may be on another thread after "await"
            }
            catch (Exception ex)
            {
                // log the error
                Console.WriteLine(ex.Message);
                Console.WriteLine($"[Server] Client {clientId} disconnected");
            }
            finally
            {
                lock (_connections)
                {
                    OnShutdown(clientId);
                    _connections.Remove(connectionTask);
                }
                tcpClient.Dispose();
            }
        }
    }
}
```

Parser.cs

```
using System.Collections.Generic;
using System.Linq;
using IMDG.Common;

namespace IMDG.Parser
```

```
{
    public class Parser
    {
        private readonly string _input;
        private int _current = 0;

        private const string Eof = "<EOF>";
        private const char Space = ' ';
        public List<string> Errors { get; } = new List<string>();

        public bool HasErrors => Errors.Any();

        public Parser(string input)
        {
            _input = input + Space + Eof;
        }

        public Command Parse()
        {
            var command = ParseCommand();
            if (command is null)
            {
                return null;
            }
            MatchEof();
            return command;
        }


        public static Command Parse(string input) => new Parser(input).Parse();

        private Command ParseCommand()
        {
            var keyword = MatchWord().ToLowerInvariant();
            switch (keyword)
            {
                case "getall":
                    return new GetAllCommand();
                case "set":
                {
                    var (key, value) = MatchKeyValue();
                    return new SetCommand(key, value);
                }
                case "get":
                {
                    var key = MatchWord();
                    return new GetCommand(key);
                }
                case "remove":
                {
                    var key = MatchWord();
                    return new RemoveCommand(key);
                }

                case "find":
                {
                    var key = MatchWord();
                    return new FindCommand(key);
                }

                case "update":
                {
                    var (key, value) = MatchKeyValue();
                    return new UpdateCommand(key, value);
                }
```

```
case "keys":
    return new GetKeysCommand();
case "clear":
    return new ClearCommand();
case "hset":
{
    var key = MatchWord();
    var field = MatchWord();
    var value = MatchWord();
    return new HSetCommand(key, field, value);
}
case "hgetall":
{
    var key = MatchWord();
    return new HGetAllCommand(key);
}
case "hget":
{
    var key = MatchWord();
    var field = MatchWord();
    return new HGetCommand(key, field);
}
case "hval":
{
    var key = MatchWord();
    return new HValCommand(key);
}
case "hremove":
{
    var key = MatchWord();
    var field = MatchWord();
    return new HRemoveCommand(key, field);
}
case "hkeys":
{
    var key = MatchWord();
    return new HKeysCommand(key);
}
case "ladd":
{
    var key = MatchWord();
    var value = MatchWord();
    return new LAddCommand(key, value);
}
case "lgetall":
{
    var key = MatchWord();
    return new LGetAllCommand(key);
}
case "lremove":
{
     var key = MatchWord();
     var field = MatchWord();
     return new LRemoveCommand(key, field);
}
case "lcount":
{
     var key = MatchWord();
     return new LCountCommand(key);
}
case Eof:
{
    return null;
}
default:
```

```
                {
                    Errors.Add("Invalid command");
                    return null;
                }
            }
        }

        private string MatchWord()
        {
            SkipWhitespace();
            var index = 0;
            while (_current + index < _input.Length && IsValidWordChar(_input[_current +
index]))
            {
                index++;
            }
            if (index == 0)
            {
                Errors.Add("Invalid command");
                return null;
            }
            var result = _input[_current..(_current + index)];
            _current += index;
            return result;

            static bool IsValidWordChar(char ch)
            {
                return char.IsLetterOrDigit(ch) || char.IsPunctuation(ch) ||
char.IsSymbol(ch);
            }
        }

        private (string Key, string Value) MatchKeyValue()
        {
            SkipWhitespace();
            var key = MatchWord();
            SkipWhitespace();
            var value = MatchWord();
            return (key, value);
        }

        private string MatchWord(string word)
        {
            var actualWord = MatchWord();
            if (actualWord == word)
            {
                return word;
            }

            Errors.Add($"expected {word}, got `{actualWord}`");
            return null;
        }


        private void MatchEof()
        {
            SkipWhitespace();
            MatchWord(Eof);
        }

        private void SkipWhitespace()
        {
            var index = 0;
            while (_current + index < _input.Length && char.IsWhiteSpace(_input[_current +
index]))
```

```
            {
                index++;
            }

            _current += index;
        }
    }
}
```

FileHandler.cs

```
using System.Collections.Generic;
using System.Diagnostics;
using System.Threading.Tasks;
using IMDG.Common;

namespace IMDG.Storage
{
    public class FileHandler
    {
        private readonly string _fileName;

        public FileHandler()
        {
            _fileName = GetFileName();
        }


        public Task UpdateFileAsync(IEnumerable<Command> commands) =>
FileHelper.TruncateAsync(_fileName, commands);


        public Task<IEnumerable<Command>> GetFile() =>
FileHelper.ReadCommandsFromFile(_fileName);

        private static string GetFileName()
        {
            var id = Process.GetCurrentProcess().Id;
            return $"StorageFile_{id}.txt";
        }
    }
}
```

Storage.cs

```
using System;
using System.Collections.Concurrent;
using System.Linq;
using System.Threading.Tasks;
using IMDG.Common;
using IMDG.Parser;

namespace IMDG.Storage
{
    public class Storage
    {
        private readonly FileHandler _fileHandler;
        private     readonly     ConcurrentDictionary<string,     Value>     _storage     =     new
ConcurrentDictionary<string, Value>();


        public Storage(FileHandler fileHandler)
        {
            _fileHandler = fileHandler;
        }
```

```
        public async Task<string> RequestAsync(Command command)
        {
            var result = command.Kind switch
            {
                CommandKind.Get => PerformGet((GetCommand) command),
                CommandKind.Set => PerformSet((SetCommand) command),
                CommandKind.GetAll => PerformGetAll((GetAllCommand) command),
                CommandKind.Keys => PerformKeys((GetKeysCommand) command),
                CommandKind.Update => PerformUpdate((UpdateCommand) command),
                CommandKind.Remove => PerformRemove((RemoveCommand) command),
                CommandKind.Find => PerformFind((FindCommand) command),
                CommandKind.Clear => PerformClear((ClearCommand) command),
                CommandKind.HSet => PerformHSet((HSetCommand) command),
                CommandKind.HGetAll => PerformHGetAll((HGetAllCommand) command),
                CommandKind.HGet => PerformHGet((HGetCommand) command),
                CommandKind.HVal => PerformHVal((HValCommand) command),
                CommandKind.HRemove => PerformHRemove((HRemoveCommand) command),
                CommandKind.HKeys => PerformHKeys((HKeysCommand) command),
                CommandKind.LAdd => PerformLAdd((LAddCommand) command),
                CommandKind.LGetAll => PerformLGetAll((LGetAllCommand) command),
                CommandKind.LRemove => PerformLRemove((LRemoveCommand) command),
                CommandKind.LCount => PerformLCount((LCountCommand) command),
                CommandKind.Replay => await PerformReplay((ReplayCommand) command),
                CommandKind.Sync => await PerformSync((SyncCommand) command),
                _ => "Invalid command"
            };
            return result;
        }

        private async Task<string> PerformReplay(ReplayCommand replayCommand)
        {
            var range = replayCommand.PartitionRange;
            var allCommands = await _fileHandler.GetFile();
            var toReplay = allCommands
                .Where(CommandFacts.CanReplay)
                .OfType<KeyCommand>()
                .Where(c => range.Contains(c.Key.GetDeterministicHashCode()));

            _storage.Clear();
            foreach (var command in toReplay) await RequestAsync(command);
            Console.WriteLine("================================");
            Console.WriteLine(PerformGetAll(new GetAllCommand()));
            Console.WriteLine("================================");
            return "Replay ok";
        }

        private async Task<string> PerformSync(SyncCommand command)
        {
            await _fileHandler.UpdateFileAsync(command.Commands);
            return "Sync Ok";
        }

        private string PerformHSet(HSetCommand command)
        {
            if (_storage.TryGetValue(command.Key, out var value))
            {
                if (value is HashValue hv)
                {
                    if (hv.Value.ContainsKey(command.Field))
                    {
                        return $"Hash map associated with key {command.Key} already have key
{command.Field} ";
                    }
                    hv.Value.TryAdd(command.Field, command.Value);
                    return $"{command.Field} {command.Value} pair has been added to hash map
associated with key {command.Key}";
                }

                return $"Type associated with key {command.Key} is not a hash-value type";
```

```
            }
            var newValue = new HashValue
            {
                Value = {[command.Field] = command.Value}
            };
            _storage[command.Key] = newValue;
            return $"{command.Field} {command.Value} pair has been added to hash map associated
with key {command.Key}";
        }

        private string PerformHGetAll(HGetAllCommand command)
        {
            if (_storage.TryGetValue(command.Key, out var value))
            {
                if (value is HashValue hv)
                {
                    return hv.Display();
                }
                return $"Type associated with key {command.Key} is not a hash-value type";
            }
            return "Key does not exists";
        }
        private string PerformHGet(HGetCommand command)
        {
            if (_storage.TryGetValue(command.Key, out var value))
            {
                if (value is HashValue hv)
                {
                    if (!hv.Value.ContainsKey(command.Value))
                    {
                        return $"No such element in database associated with key {command.Key}";
                    }
                    return hv.Value[command.Value];
                }
                return $"Type associated with key {command.Key} is not a hash-value type";
            }
            return "Key does not exists";
        }

        private string PerformHVal(HValCommand command)
        {
            if (_storage.TryGetValue(command.Key, out var value))
            {
                if (value is HashValue hv)
                {
                    return hv.GetValues();
                }
                return $"Type associated with key {command.Key} is not a hash-value type";
            }
            return "Key does not exists";
        }

        private string PerformHKeys(HKeysCommand command)
        {
            if (_storage.TryGetValue(command.Key, out var value))
            {
                if (value is HashValue hv)
                {
                    return hv.GetKeys();
                }
                return $"Type associated with key {command.Key} is not a hash-value type";
            }
            return "Key does not exists";
        }
        private string PerformHRemove(HRemoveCommand command)
        {
            if (_storage.TryGetValue(command.Key, out var value))
            {
                if (value is HashValue hv)
```

```
                {
                    if (hv.Value.ContainsKey(command.Value))
                    {
                        hv.Value.TryRemove(command.Value, out var answer);
                        return answer + " " + "removed";
                    }
                    return $"No such element in database associated with key {command.Value}";
                }
                return $"Type associated with key {command.Key} is not a hash-value type";
            }
            return "Key does not exists";
        }
        private string PerformLAdd(LAddCommand command)
        {
            if (_storage.TryGetValue(command.Key, out var value))
            {
                if (value is ListValue lv)
                {
                    if (lv.Value.Contains(command.Value))
                    {
                        return $"List associated with key {command.Key} already have element
{command.Value} ";
                    }
                    lv.Value.Add(command.Value);
                    return $"{command.Value} has been added to list associated with key
{command.Key}";
                }

                return $"Type associated with key {command.Key} is not a hash-value type";
            }
            var newValue = new ListValue
            {
                Value = { command.Value }
            };
            _storage[command.Key] = newValue;
            return $"{command.Value} has been added to list associated with key {command.Key}";
        }
        private string PerformLGetAll(LGetAllCommand command)
        {
            if (_storage.TryGetValue(command.Key, out var value))
            {
                if (value is ListValue lv)
                {
                    return lv.Display();
                }
                return $"Type associated with key {command.Key} is not a list type";
            }
            return "Key does not exists";
        }
        private string PerformLRemove(LRemoveCommand command)
        {
            if (_storage.TryGetValue(command.Key, out var value))
            {
                if (value is ListValue lv)
                {
                    if (lv.Value.Contains(command.Value))
                    {
                        lv.Value.Remove(command.Value);
                        return $"{command.Value} has been removed from list associated with key
{command.Key}";
                    }
                    return $"List associated with key {command.Key} don't have element
{command.Value}";
                }
                return $"Type associated with key {command.Key} is not a list type";
            }
            return "Key does not exists";
        }
        private string PerformLCount(LCountCommand command)
```

```
        {
            if (_storage.TryGetValue(command.Key, out var value))
            {
                if (value is ListValue lv)
                {
                    return lv.Value.Count().ToString();
                }
                return $"Type associated with key {command.Key} is not a list type";
            }
            return "Key does not exists";
        }
        private string PerformSet(SetCommand command)
        {
            if (_storage.TryAdd(command.Key, new StringValue(command.Value)))
            {
                return command.Value;
            }

            // TODO: add error handling
            return "Key already exists";
        }

        private string PerformGetAll(GetAllCommand command)
        {
            if (!_storage.IsEmpty)
            {
                return _storage.Aggregate("", (current, keyValue) => current + ($"\nKey:
{keyValue.Key}" + " " + $"Value: {keyValue.Value.Display()}"));
            }

            // TODO: add error handling
            return "The storage is empty";
        }
        private string PerformGet(GetCommand command)
        {
            if (_storage.TryGetValue(command.Key, out var value))
            {   if (value is StringValue)
                {
                    return value.Display();
                }
                return $"Type associated with key {command.Key} is not a common type";
            }
            return "Key does not exits";
        }
        private string PerformKeys(GetKeysCommand command)
        {
            if (!_storage.IsEmpty)
            {
                return _storage.Aggregate("", (current, keyValue) => ($"\n{keyValue.Key}"));
            }

            return "The storage is empty";
        }
        private string PerformUpdate(UpdateCommand command)
        {
            if (_storage.TryGetValue(command.Key, out var value))
            {
                if (value is StringValue)
                {
                    _storage.TryUpdate(command.Key,        new        StringValue(command.Value),
_storage[command.Key]);
                    return "Updated";
                }
                return $"Type associated with key {command.Key} is not a common type";
            }
            return "No such element in database";
        }
        private string PerformRemove(RemoveCommand command)
        {
```

```
            if (_storage.ContainsKey(command.Key))
            {
                _storage.TryRemove(command.Key, out var answer);
                return answer.Display() + " " + "removed";
            }

            return "No such element in database";
        }
        private string PerformFind(FindCommand command)
        {
            if (_storage.ContainsKey(command.Key))
            {
                return $"The element with key {command.Key} is in database";
            }

            return "No such element in database";
        }
        private string PerformClear(ClearCommand command)
        {
            _storage.Clear();

            return "Database has been cleared";
        }
    }
}
```

StorageMain.cs

```
using System;
using System.Buffers;
using System.IO;
using System.Net.Sockets;
using System.Text;
using System.Threading.Tasks;
using IMDG.Common;
using Newtonsoft.Json;

namespace IMDG.Storage
{
    public static class StorageMain
    {
        private static async Task Main(string[] args)
        {
            var fh = new FileHandler();
            var storage = new Storage(fh);
            const int port = 7001;
            var pool = MemoryPool<byte>.Shared;
            using var byteRead = pool.Rent(100 * 1024);
            using var byteWrite = pool.Rent(100 * 1024);
            try
            {
                using var client = new TcpClient("127.0.0.1", port);
                await using var stream = client.GetStream();
                while (true)
                {
                    var request = await GetRequest(stream, byteRead.Memory);
                    if (request == null) continue;
                    Console.WriteLine($"[Storage] got {request.Kind}");
                    var answer =  await CheckAndPerformRequest(request, storage);
                    Console.WriteLine($"[Storage] Send back {answer}");
                    await SendAnswer(stream, answer, byteWrite.Memory);
                }

                //client.Close();
                //Console.WriteLine("Client closed the connection");
            }
            catch (Exception e)
```

```
            {
                Console.WriteLine(e.Message);
                throw;
            }

        }
        private static async Task<Command> GetRequest(NetworkStream stream, Memory<byte>
byteRead)
        {
            var length = await stream.ReadAsync(byteRead);
            var request = Encoding.UTF8.GetString(byteRead.Span.Slice(0, length));
            Console.WriteLine(request);
            return JsonConvert.DeserializeObject<Command>(request, new
JsonSerializerSettings
            {
                TypeNameHandling = TypeNameHandling.Objects
            });
        }

        private static async Task SendAnswer(Stream stream, string answer, Memory<byte>
byteWrite)
        {
            var len = Encoding.UTF8.GetBytes(answer, byteWrite.Span);
            await stream.WriteAsync(byteWrite.Slice(0, len));
            await stream.FlushAsync();
        }
        private static async Task<string> CheckAndPerformRequest(Command request, Storage
storage)
        {
            if (request is null)
            {
                // never happens
                return "Internal error";
            }

            return await storage.RequestAsync(request);
        }
    }
}
```